RSAC 2025 Conference

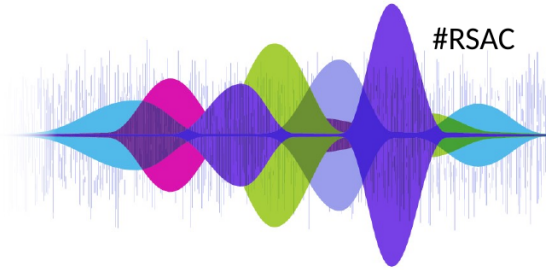
Many Voices.
One Community.

SESSION ID: USENIX-M03A

HYPERPILL: Fuzzing for Hypervisor-bugs by Leveraging the Hardware Virtualization Interface

Alexander Bulekov

Security Researcher
http://www.linkedin.com/in/a1xndr



Disclaimer

Presentations are intended for educational purposes only and do not replace independent professional judgment. Statements of fact and opinions expressed are those of the presenters individually and, unless expressly stated to the contrary, are not the opinion or position of RSA Conference LLC or any other co-sponsors. RSA Conference LLC does not endorse or approve, and assumes no responsibility for, the content, accuracy or completeness of the information presented.

Attendees should note that sessions may be audio- or video-recorded and may be published in various media, including print, audio and video formats without further notice. The presentation template and any media capture are subject to copyright protection.

© 2025 RSA Conference LLC or its affiliates. The RSAC and RSAC CONFERENCE logos and other trademarks are proprietary. All rights reserved.

The views and opinions expressed in this presentation are my own and do not necessarily reflect those of my employer

HyperPill

Fuzzing for Hypervisor-bugs by leveraging the Hardware Virtualization Interface

Alexander Bulekov — EPFL
Qiang Liu — Boston University
Manuel Egele — Zhejiang University
Mathias Payer

HyperPill: Fuzzing for Hypervisor-bugs by Leveraging the Hardware Virtualization Interface

Alexander Bulekov[*†§] alxndr@bu.edu — *Qiang Liu*[*‡] qiang.liu@epfl.ch — *Manuel Egele*[†] megele@bu.edu — *Mathias Payer*[*] mathias.payer@nebelwelt.net

[*]EPFL [†]Boston University [‡]Zhejiang University [§]Amazon

Abstract

The security guarantees of cloud computing depend on the isolation guarantees of the underlying hypervisors. Prior works have presented effective methods for automatically identifying vulnerabilities in hypervisors. However, these approaches are limited in scope. For instance, their implementation is typically hypervisor-specific and limited by requirements for detailed grammars, access to source-code, and assumptions about hypervisor behaviors. In practice, complex closed-source and recent open-source hypervisors are often not suitable for off-the-shelf fuzzing techniques.

HyperPill introduces a generic approach for fuzzing arbitrary hypervisors. HyperPill leverages the insight that although hypervisor implementations are diverse, all hypervisors rely on the identical underlying hardware-virtualization interface to manage virtual-machines. To take advantage of the hardware-virtualization interface, HyperPill makes a snapshot of the hypervisor, inspects the snapshotted hardware state to enumerate the hypervisor's input-spaces, and leverages feedback-guided snapshot-fuzzing within an emulated environment to identify vulnerabilities in arbitrary hypervisors. In our evaluation, we found that beyond being the first hypervisor-fuzzer capable of identifying vulnerabilities in arbitrary hypervisors across all major attack-surfaces (i.e., PIO/MMIO/Hypercalls/DMA), HyperPill also outperforms state-of-the-art approaches that rely on access to source-code, due to the granularity of feedback provided by HyperPill's emulation-based approach. In terms of coverage, HyperPill outperformed past fuzzers for 10/12 QEMU devices, without the API hooking or source-code instrumentation techniques required by prior works. HyperPill identified 26 new bugs in recent versions of QEMU, Hyper-V, and macOS Virtualization Framework across four device-categories.

All work was completed prior to author joining Amazon.

1 Introduction

Hypervisors provide the security foundations necessary for the cloud. They enable efficient use of hardware resources, by colocating workloads from multiple tenants on the same bare-metal machines, isolated in individual virtual-machines (VMs). As such, hypervisors ensure that code running in VMs cannot violate the virtualization boundary (e.g., by performing a VM escape attack) and compromise the workloads of the other tenants, or the hypervisor itself.

Unfortunately, VM escape attacks are a tangible reality. Hundreds of bugs have been identified in the complex hypervisor code. Due to the severity of these bugs, hypervisor compromises are awarded large bug bounties, similar to other high-value targets such as web browsers and mobile devices [53]. In parallel, fuzzing has emerged as one of the most powerful techniques for automatically uncovering vulnerabilities in a large range of software [4,10,14,18,20,21,27,28,39,45,49,52]. As such, a significant amount of academic research has focused on leveraging fuzzing to automatically identify bugs in hypervisor code, so that they can be promptly fixed, preventing malicious exploitation [6,13,26,29,32,37,38].

State-of-the-art approaches [6,26] are capable of automatically finding complex bugs across most major attack-surfaces (i.e., PIO/MMIO/DMA). However, these approaches rely on access and manual modifications to hypervisor source-code to effectively fuzz virtual-devices. Even with access to source-code, porting current methods to new targets is a non-trivial process that requires considerable manual effort by an expert. Furthermore, most fuzzers do not handle the hypercall attack-surface as hypercalls are often implemented in a separate component from the core device-emulation (e.g., in the OS kernel), for performance reasons. Thus, even though major open-source targets such as QEMU and VirtualBox have been extensively fuzzed, closed-source targets such as Hyper-V and macOS Virtualization Framework have not been thoroughly fuzzed with state-of-the-art methods.

Due to the diversity of designs and runtimes (see Tab. 1), harnessing a new hypervisor requires a combination of knowl-

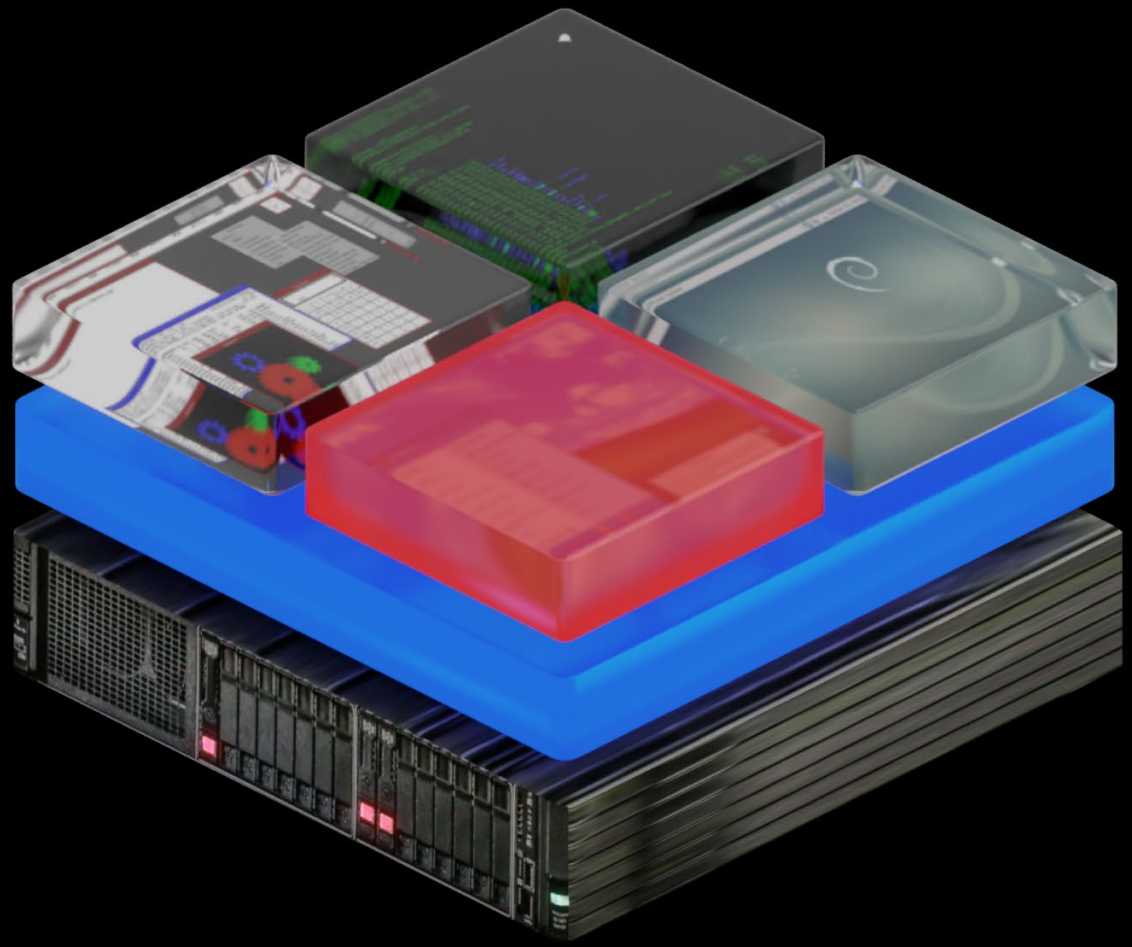
hexhive

BOSTON UNIVERSITY

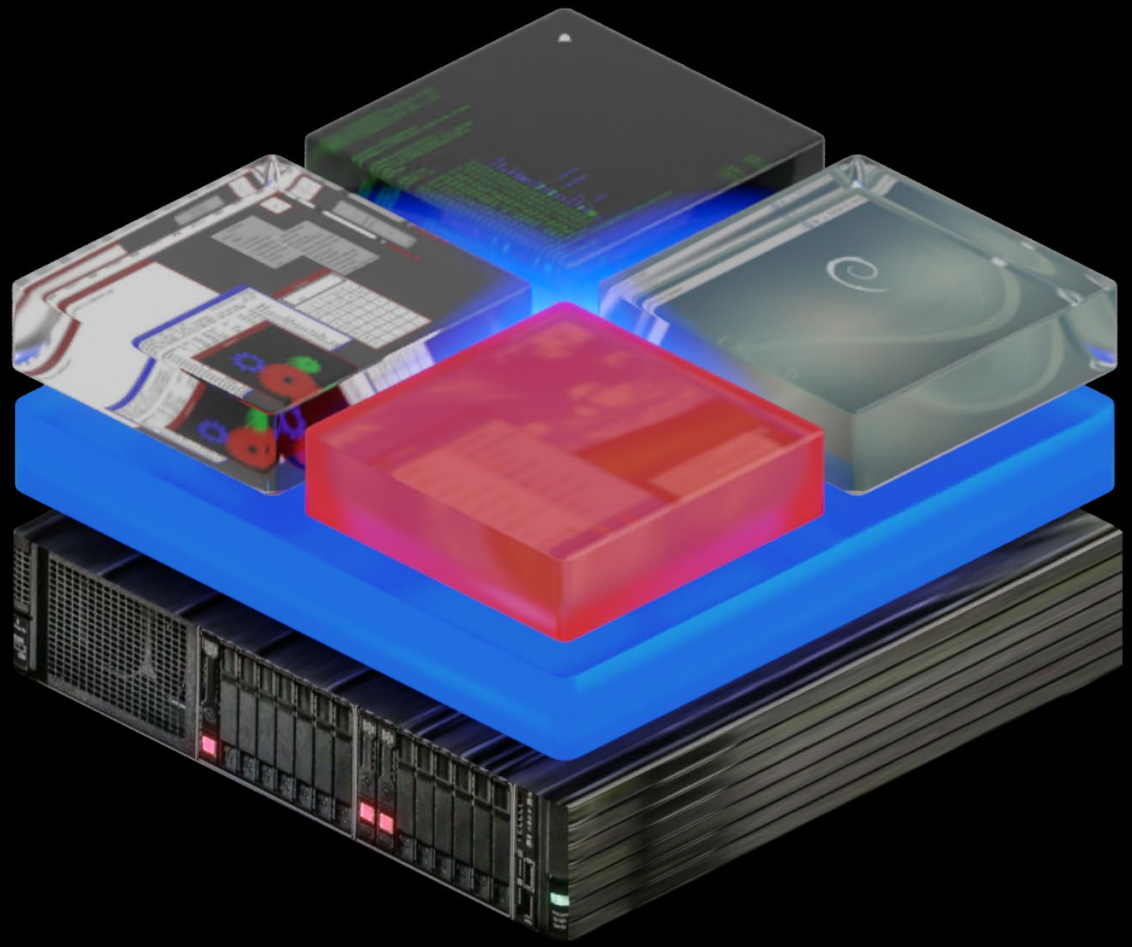
SeclaBU

Motivation

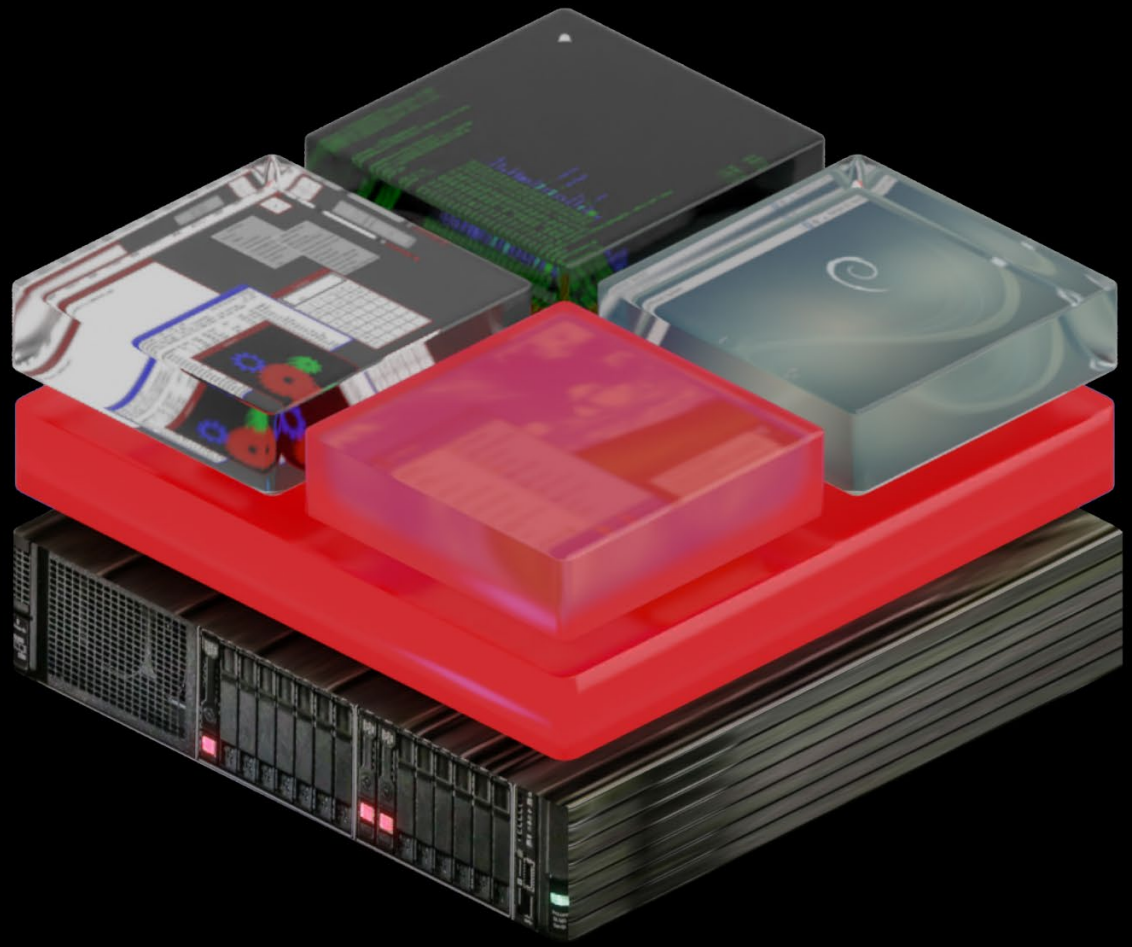
Motivation



Motivation



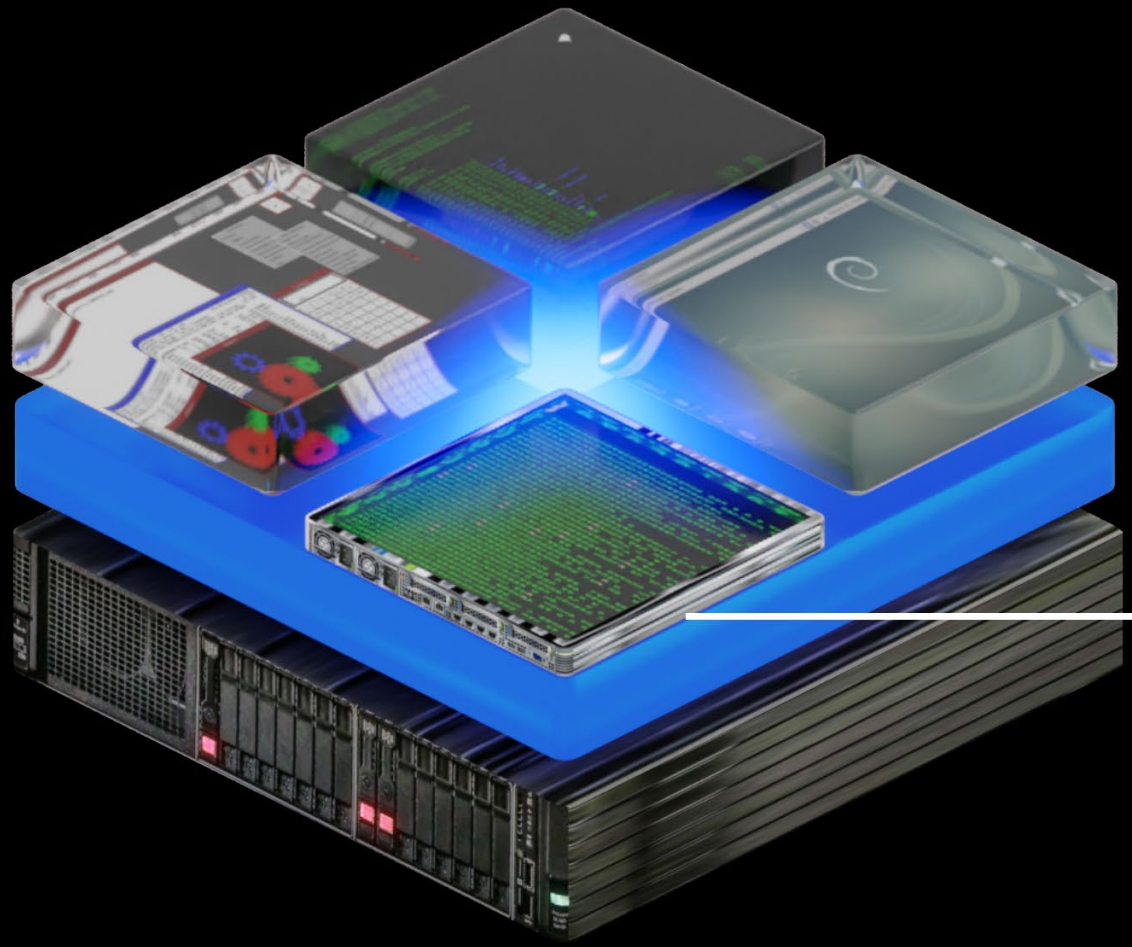
Motivation



Motivation

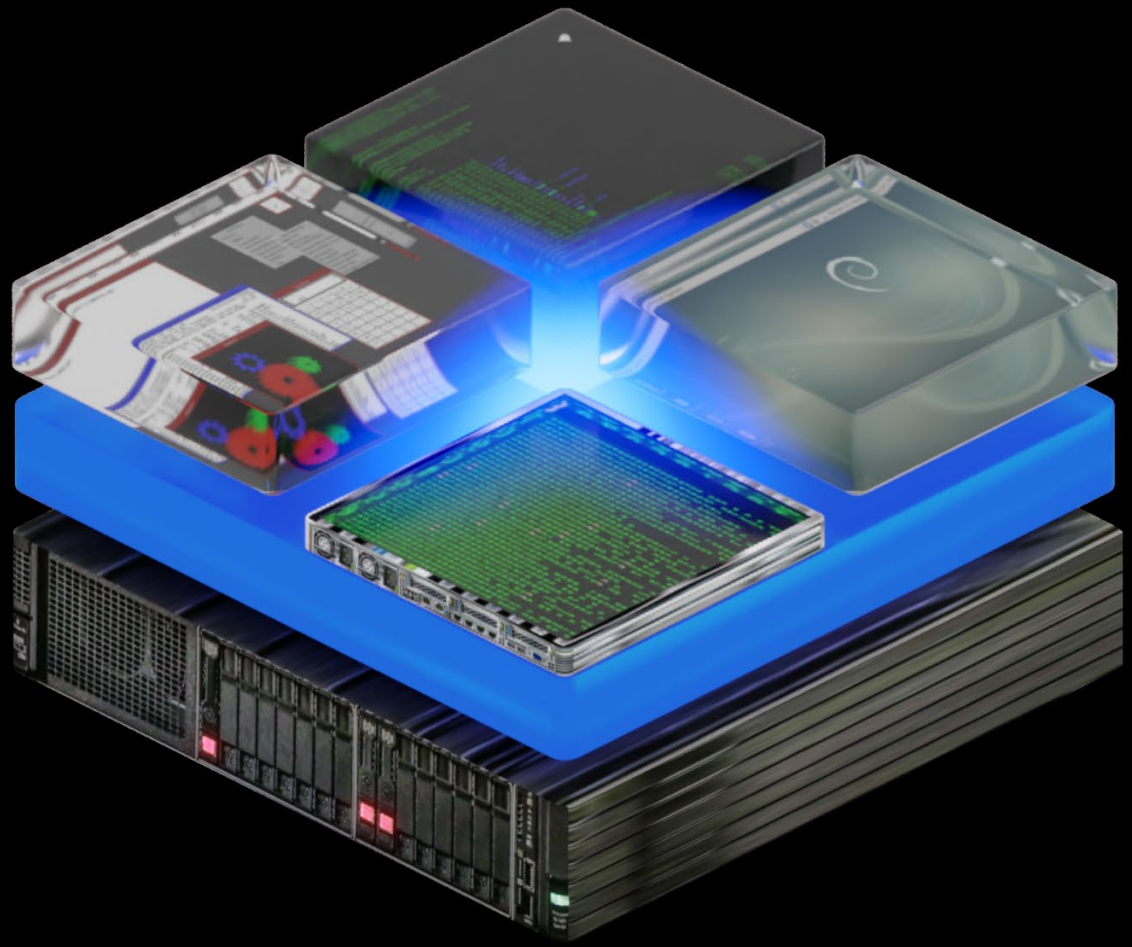
Motivation

Motivation



Virtualization Interface

Motivation



Applications:

Cloud
Personal Computing
Development
Security Research
Mobile
Automotive

Fuzzing Hypervisors: **Challenges**

Hypervisor inputs feature complex semantics

Fuzzing Hypervisors: **Challenges**

Hypervisor inputs feature complex semantics

There are many hypervisors

Fuzzing Hypervisors: **Challenges**

Hypervisor inputs feature complex semantics

There are many hypervisors

Implementations are diverse

Fuzzing Hypervisors: **Challenges**

Hypervisor inputs feature complex semantics

There are many hypervisors

Implementations are diverse

Hypervisors are low-level systems software

Fuzzing Hypervisors: **Challenges**

Hypervisor inputs feature complex semantics

There are many hypervisors

Implementations are diverse

Hypervisors are low-level systems software

Hypervisors rely on sophisticated CPU features

Fuzzing Hypervisors: **Challenges**

Hypervisor inputs feature complex semantics

There are many hypervisors

Implementations are diverse

Hypervisors are low-level systems software

All Hypervisors use an identical CPU virtualization interface

The Wizard, the Watering Robot, and the Seeds

The Input Space

What interfaces do hypervisors expose to VMs?

The Input Space

What interfaces do hypervisors expose to VMs?

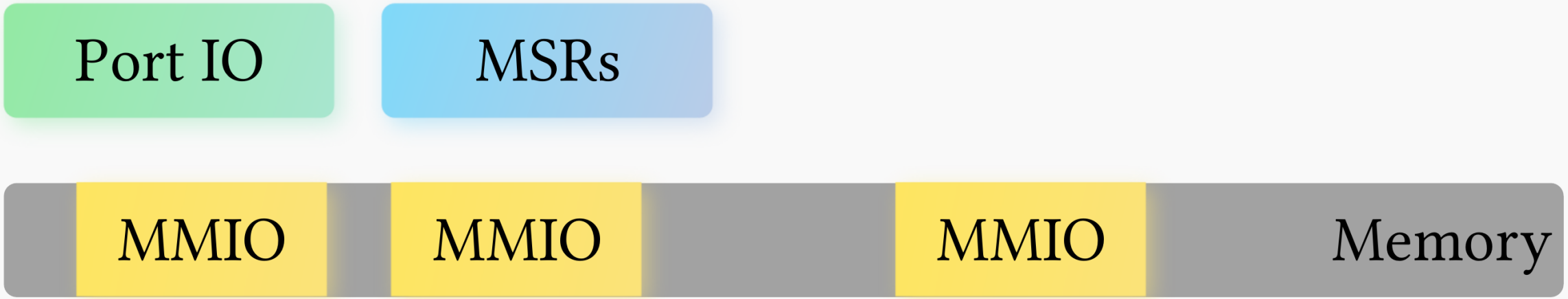
Port IO

MSRs

Memory

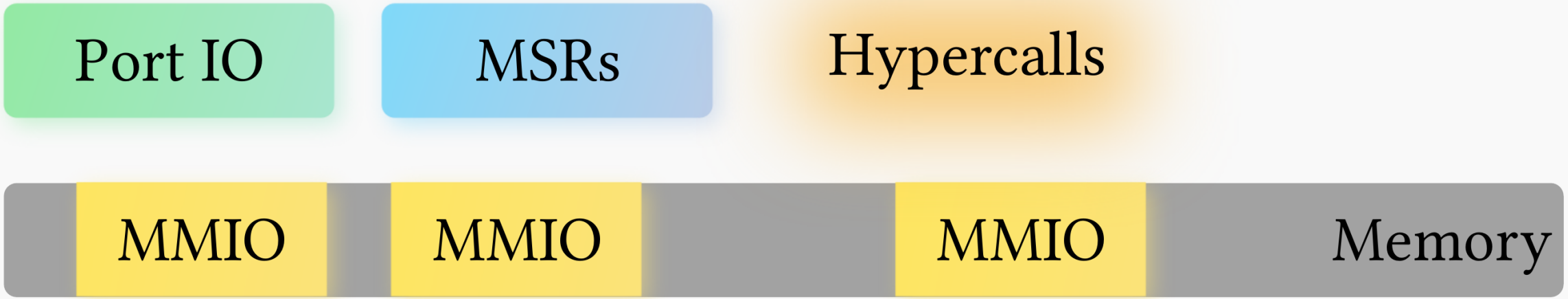
The Input Space

What interfaces do hypervisors expose to VMs?



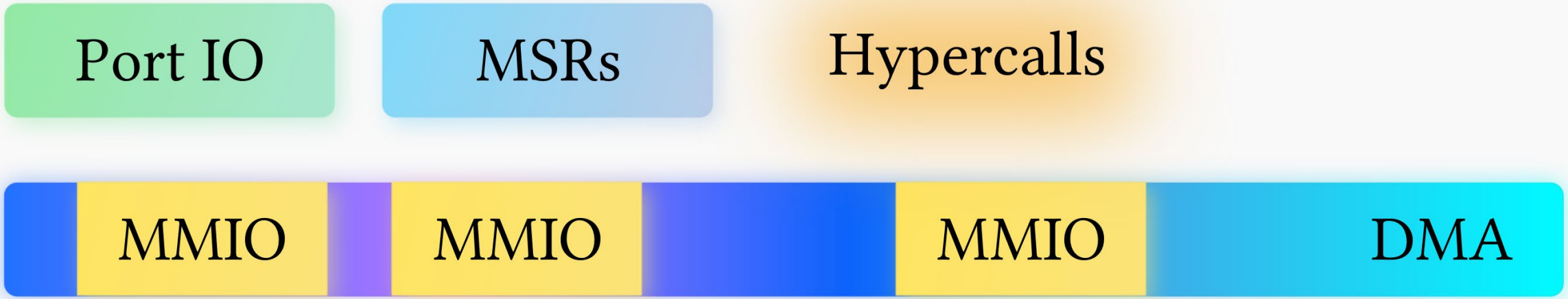
The Input Space

What interfaces do hypervisors expose to VMs?



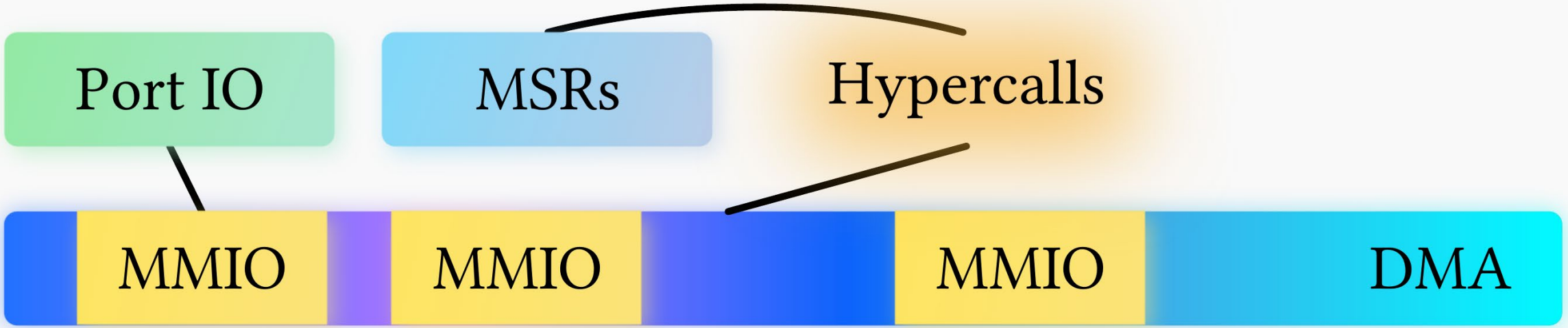
The Input Space

What interfaces do hypervisors expose to VMs?



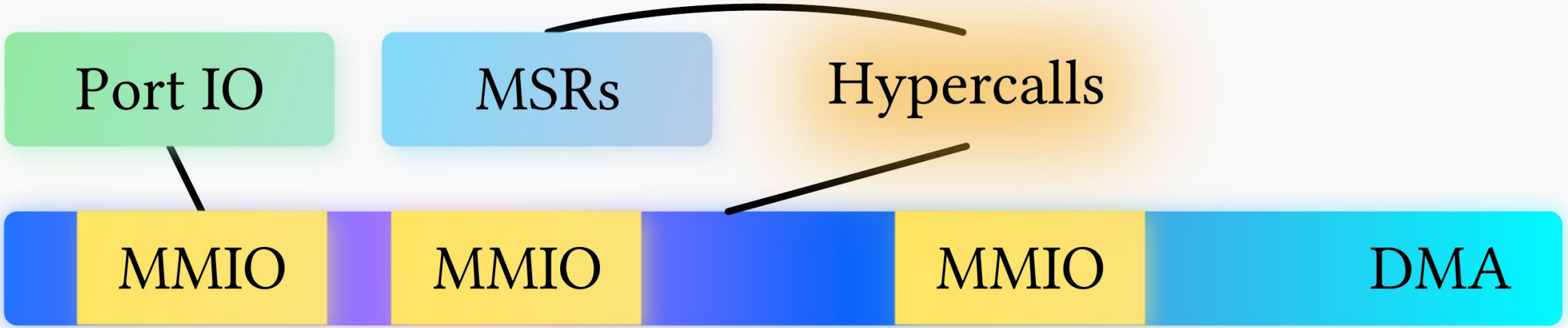
The Input Space

What interfaces do hypervisors expose to VMs?



The Input Space

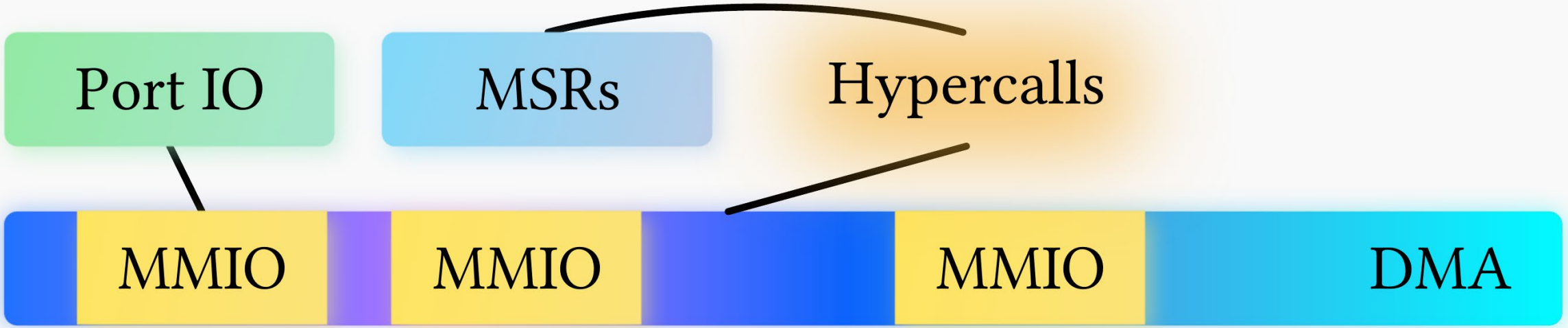
What interfaces do hypervisors expose to VMs?



The combined hypervisor input space is enormous

The Input Space

What interfaces do hypervisors expose to VMs?



The combined hypervisor input space is enormous

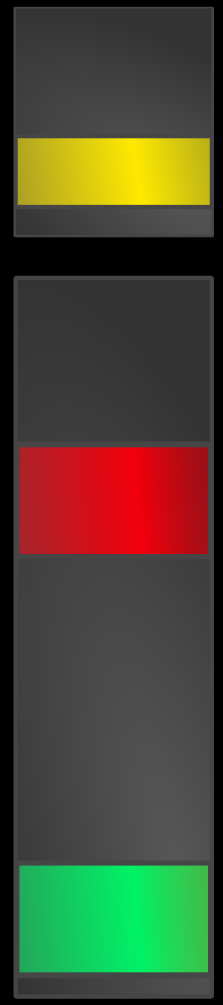
Hypervisor semantics are complex and variable

Input Semantics

```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```

Input Semantics

```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```



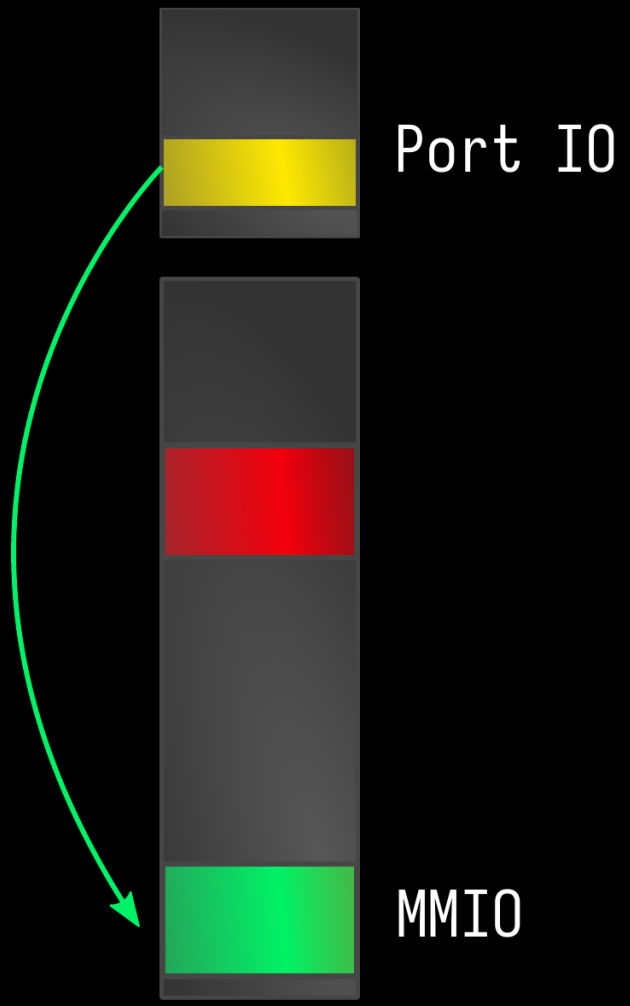
Port IO

Input Semantics

```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```

Specifies Addr of

Creates



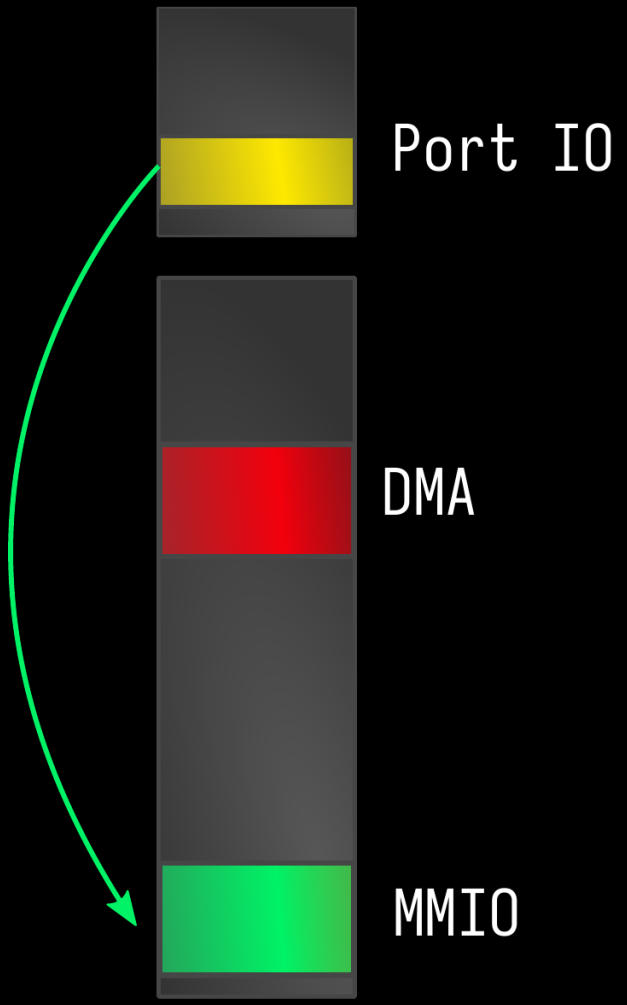
Input Semantics

```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```

Specifies Addr of

Creates

Specifies Addr of



Input Semantics

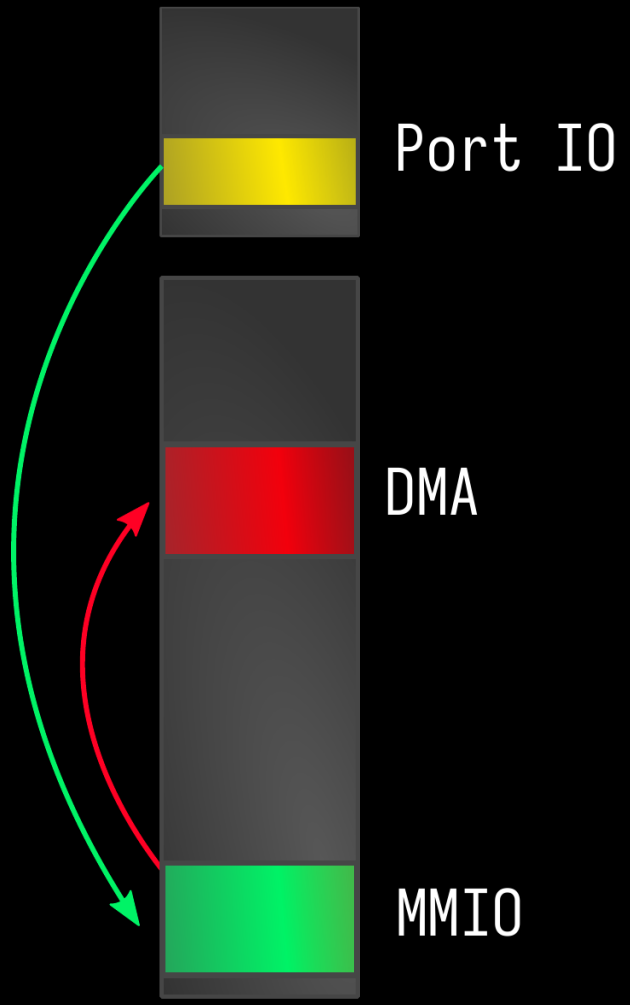
```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```

Specifies Addr of

Creates

Specifies Addr of

Causes Access to



Input Semantics

```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```

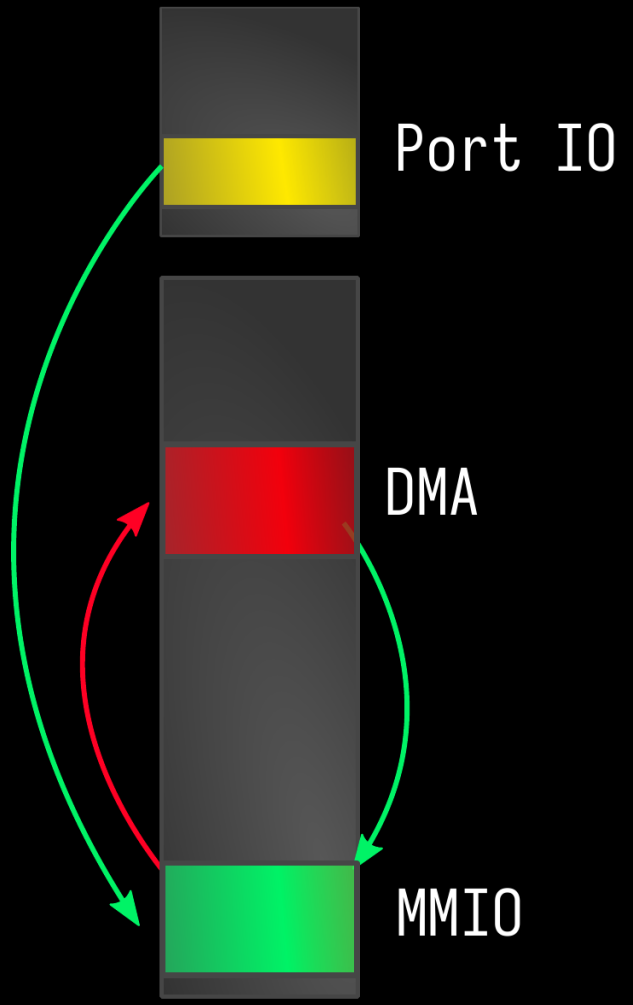
Specifies Addr of

Creates

Specifies Addr of

References

Causes Access to



Input Semantics

```
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0xff
write 0xe1020118 0xffffffff
write 0xe1020420 0xffffffff
write 0xe1020424 0xffffffff
write 0xe102042b 0xff
write 0xe1020429 0x5 0x0015c5e5c0
write 0x5c041 0x0402e1
write 0x5c048 0x8a
write 0x5c04a 0x31
write 0x5c04b 0xff
write 0xe1020403 0xff
```

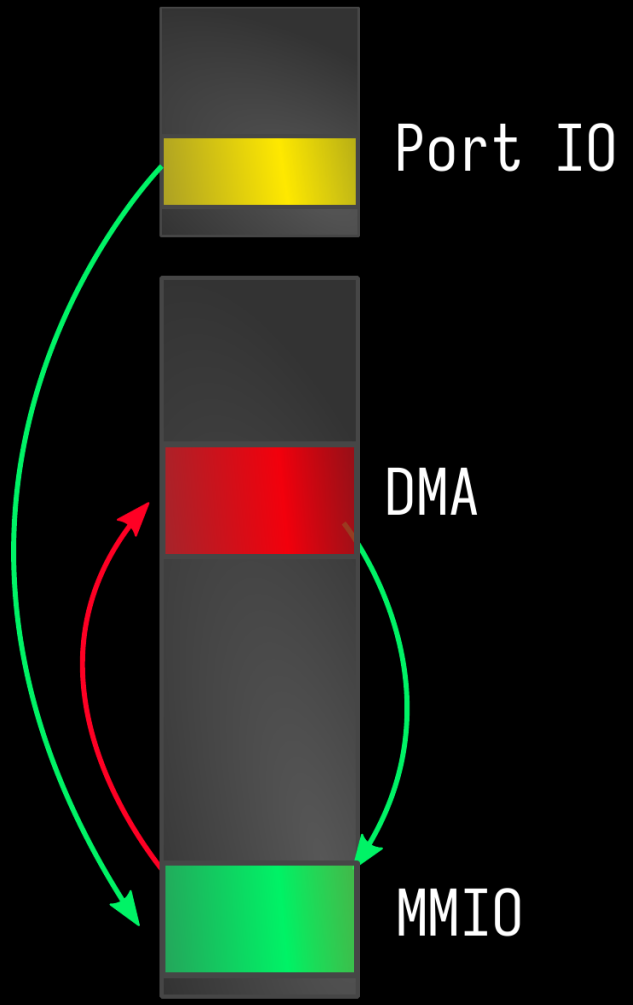
Specifies Addr of

Creates

Specifies Addr of

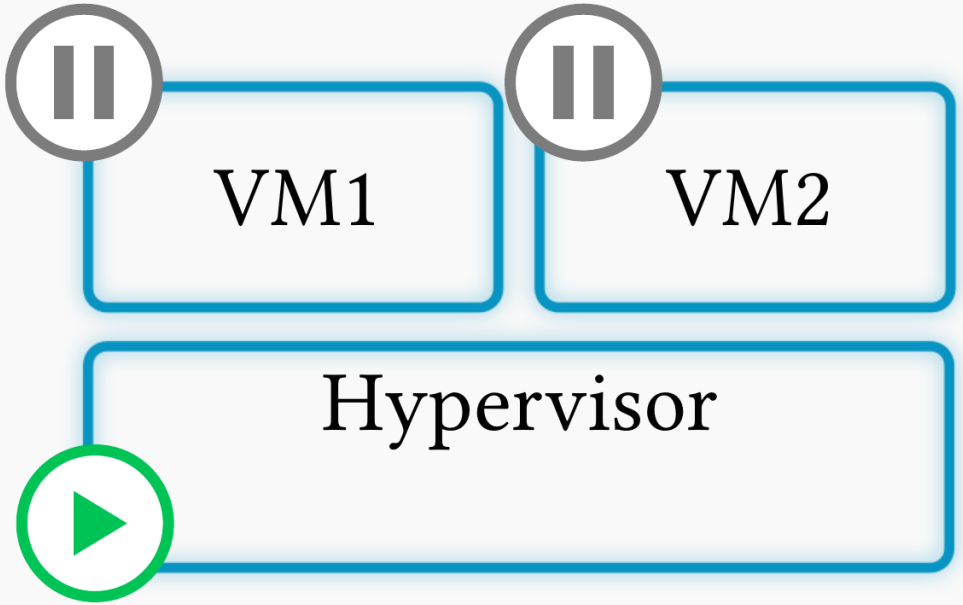
References

Causes Access to

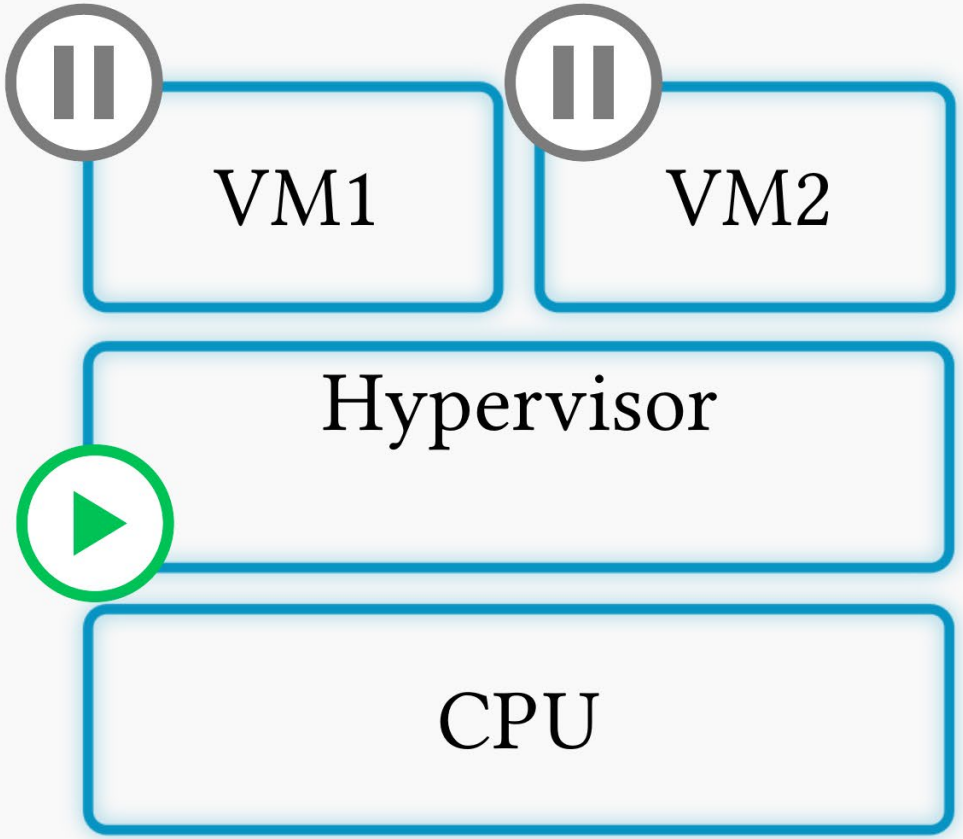


Inputs can interact with virtually any address

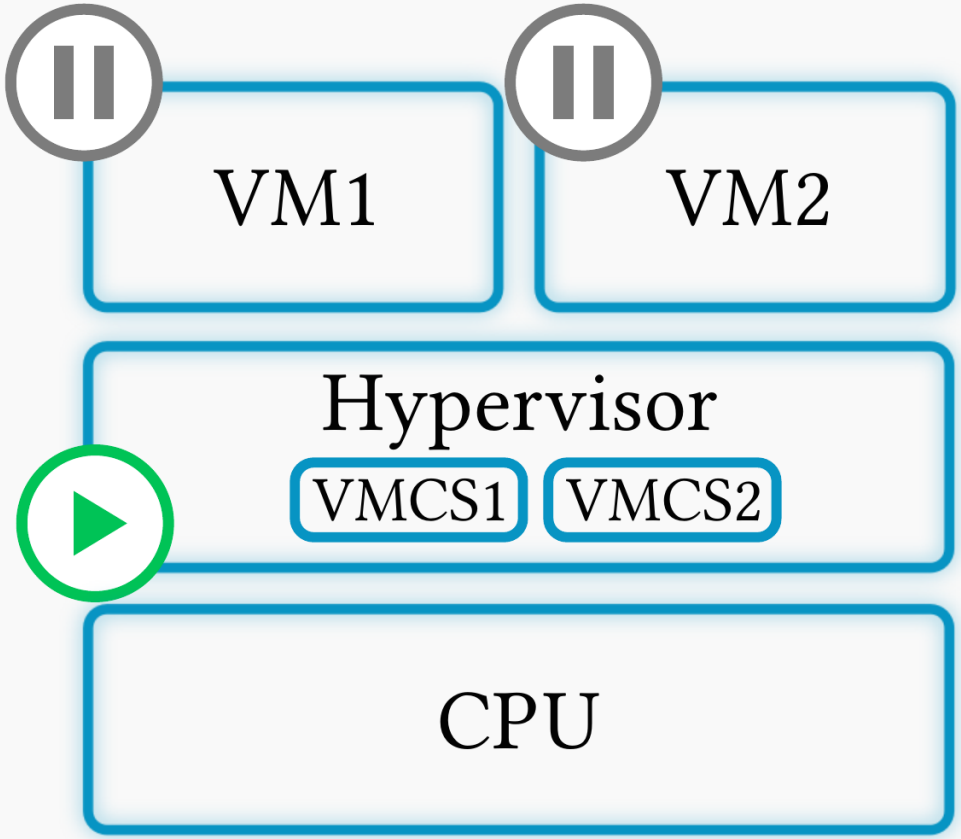
The Hardware Virtualization Interface



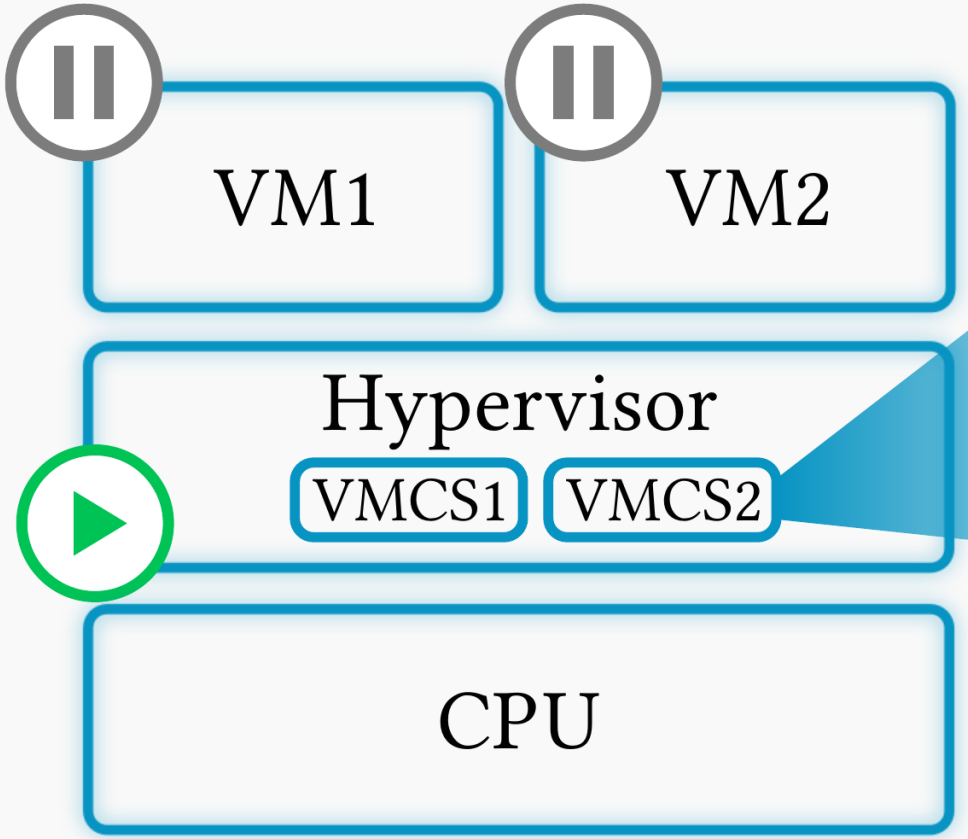
The Hardware Virtualization Interface



The Hardware Virtualization Interface

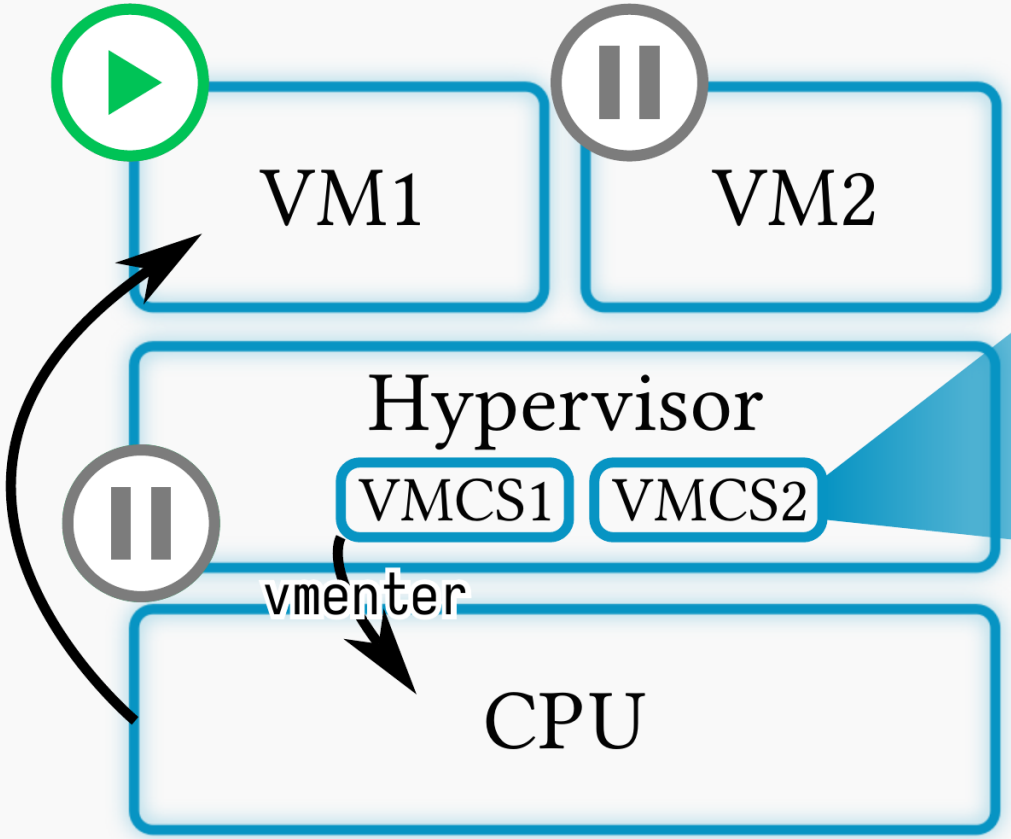


The Hardware Virtualization Interface



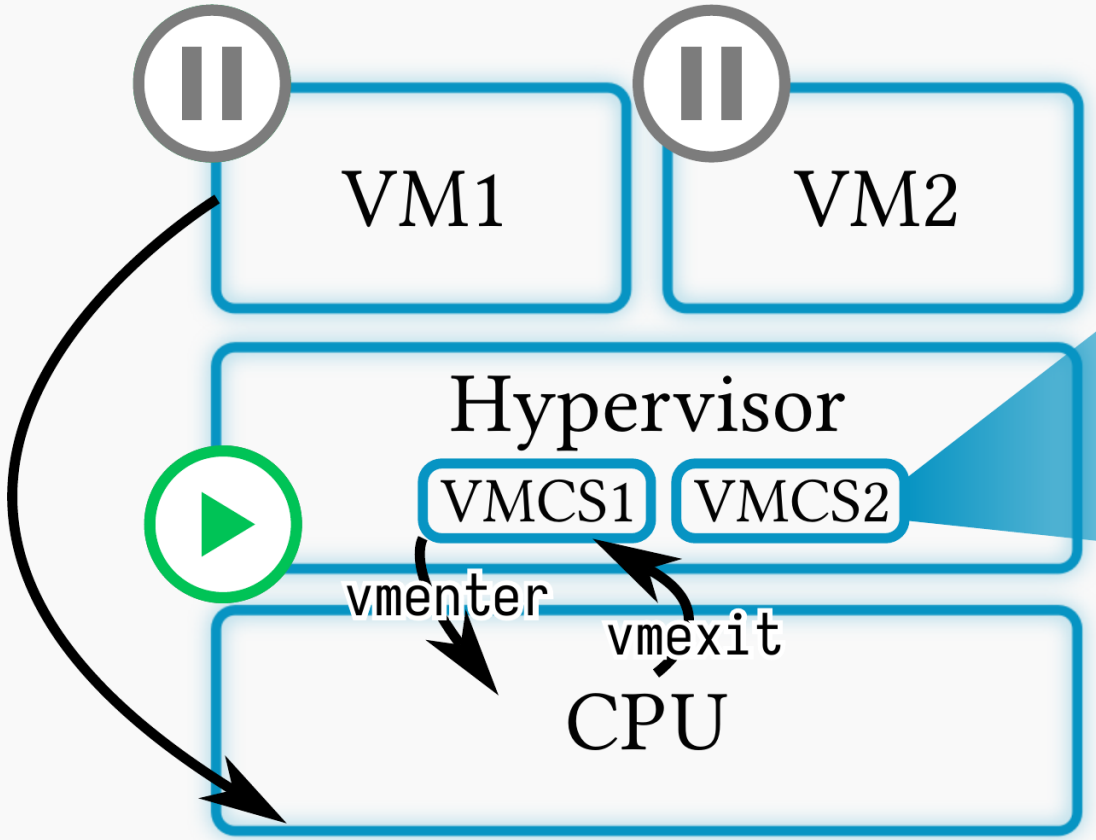
- Guest vCPU State
- Memory allocated to the Guest
- MMIO Regions
- PIO Regions
- Enabled MSRs
- VM Exit Reasons

The Hardware Virtualization Interface



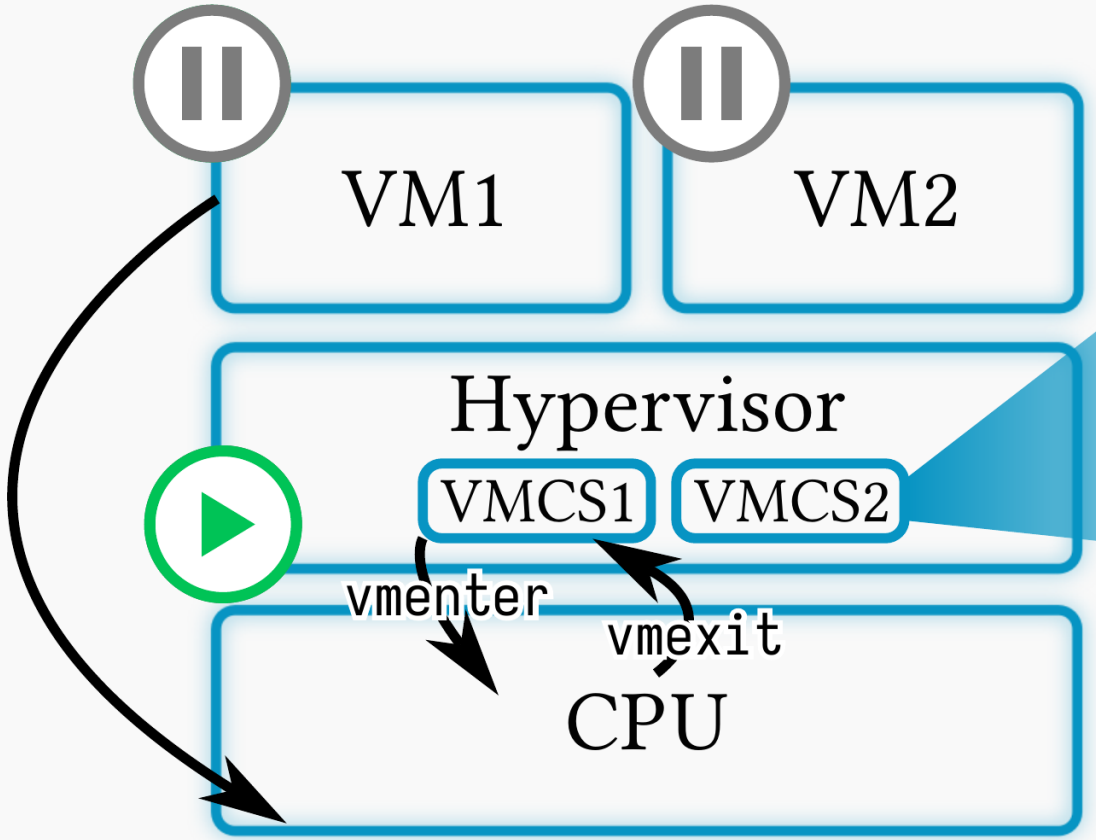
- Guest vCPU State
- Memory allocated to the Guest
- MMIO Regions
- PIO Regions
- Enabled MSRs
- VM Exit Reasons

The Hardware Virtualization Interface



- Guest vCPU State
- Memory allocated to the Guest
- MMIO Regions
- PIO Regions
- Enabled MSRs
- VM Exit Reasons

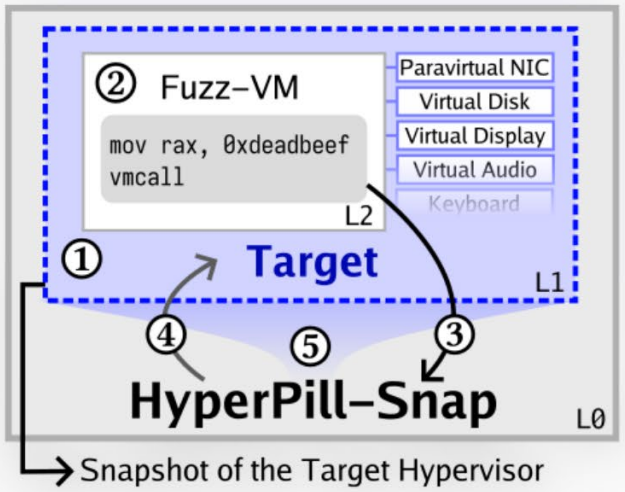
The Hardware Virtualization Interface

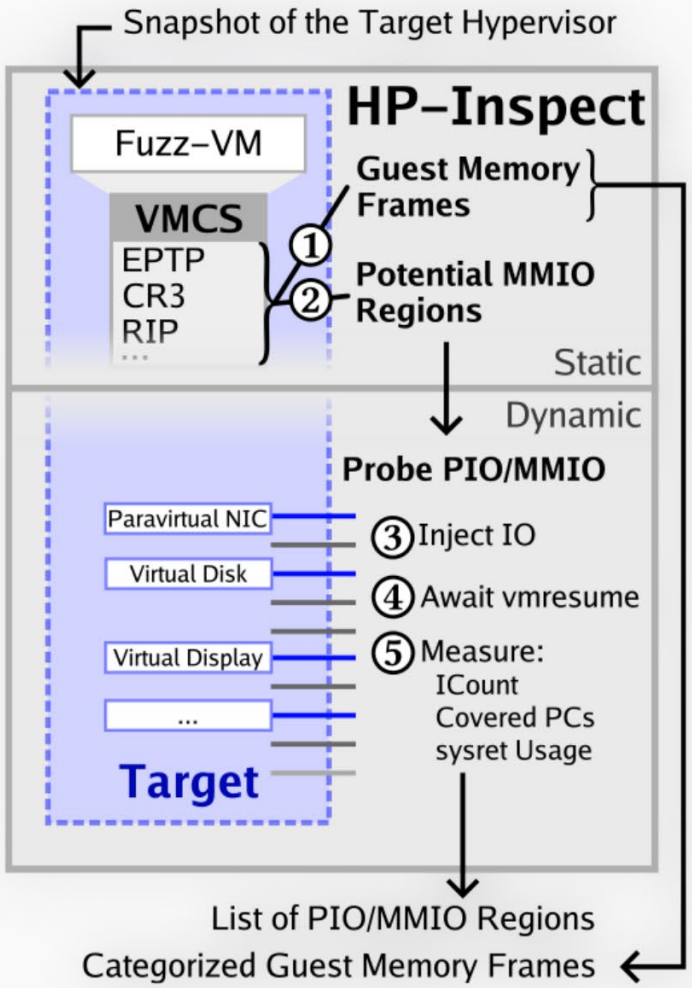


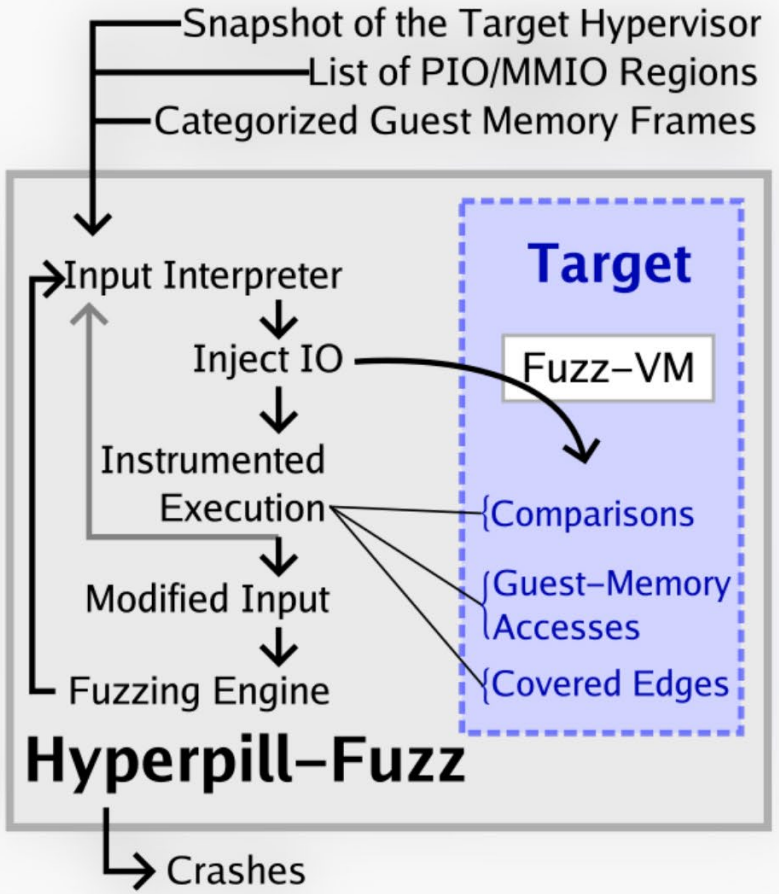
- Guest vCPU State
- Memory allocated to the Guest
- MMIO Regions
- PIO Regions
- Enabled MSRs
- VM Exit Reasons

Every Hypervisor must use an identical CPU virtualization interface.
HyperPill leverages this fact to fuzz any hypervisor - generically.

HyperPill

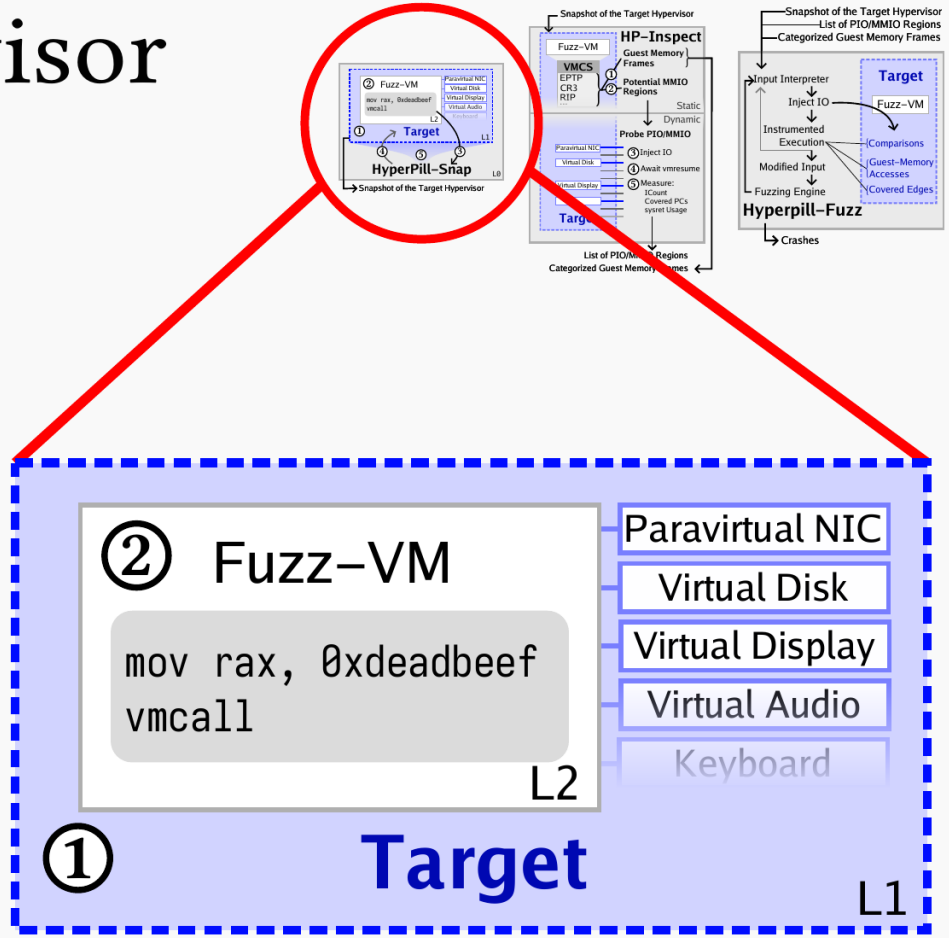

② Fuzz-VM
mov rax, 0xdeadbeef
vmcall
L2
Paravirtual NIC
Virtual Disk
Virtual Display
Virtual Audio
Keyboard
①
Target
L1
④
⑤
③
HyperPill-Snap
L0
Snapshot of the Target Hypervisor


Snapshot of the Target Hypervisor
HP-Inspect
Fuzz-VM
VMCS
EPTP
CR3
RIP
...
① Guest Memory Frames
② Potential MMIO Regions
Static
Dynamic
Probe PIO/MMIO
Paravirtual NIC
Virtual Disk
Virtual Display
...
Target
③ Inject IO
④ Await vmresume
⑤ Measure:
ICount
Covered PCs
sysret Usage
List of PIO/MMIO Regions
Categorized Guest Memory Frames


Snapshot of the Target Hypervisor
List of PIO/MMIO Regions
Categorized Guest Memory Frames
Input Interpreter
Inject IO
Instrumented Execution
Modified Input
Fuzzing Engine
Target
Fuzz-VM
Comparisons
Guest-Memory Accesses
Covered Edges
Hyperpill-Fuzz
Crashes

Step 1: Snapshotting the Hypervisor

Problem: Make a snapshot of the hypervisor just as it is about to handle a VM Exit.

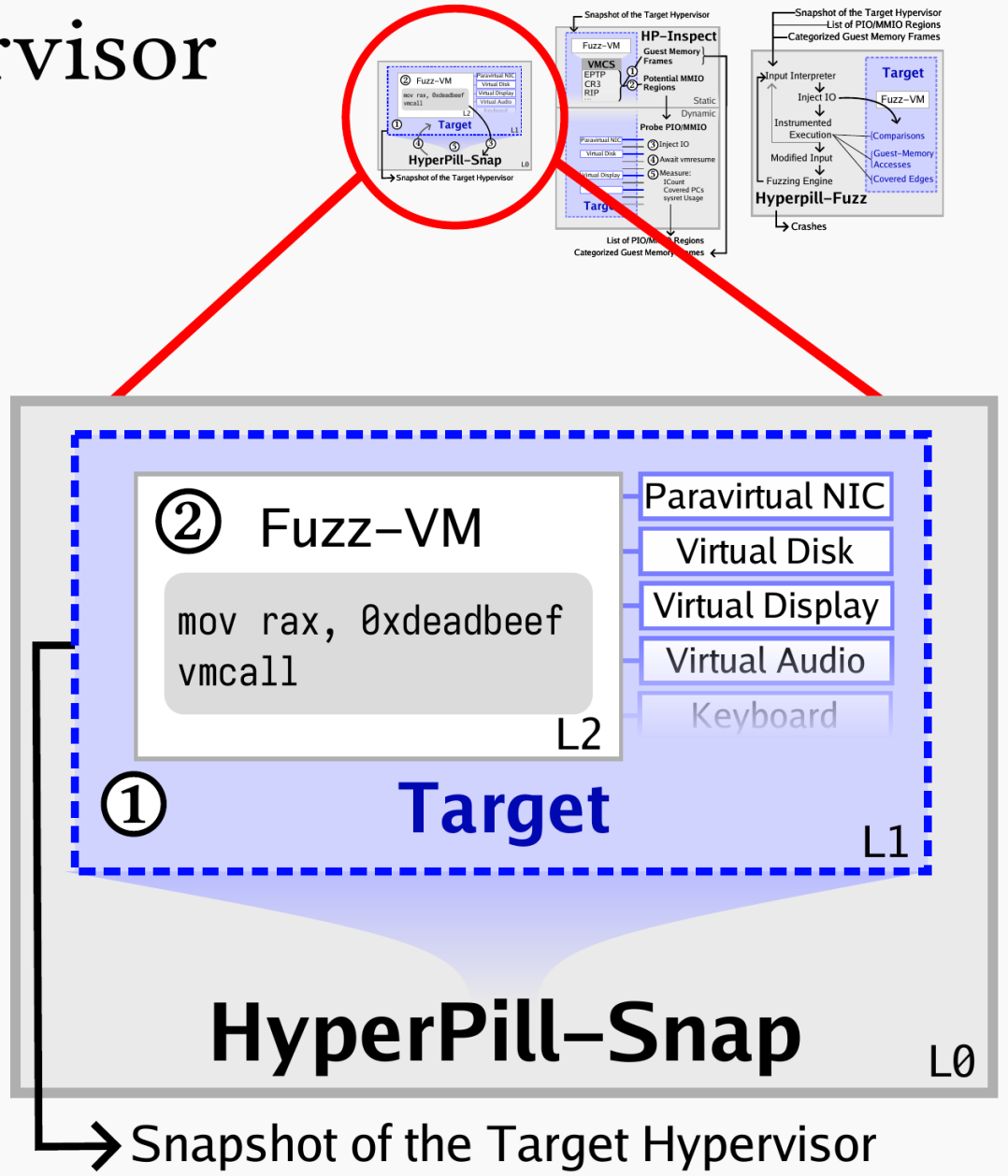


Step 1: Snapshotting the Hypervisor

Problem: Make a snapshot of the hypervisor just as it is about to handle a VM Exit.

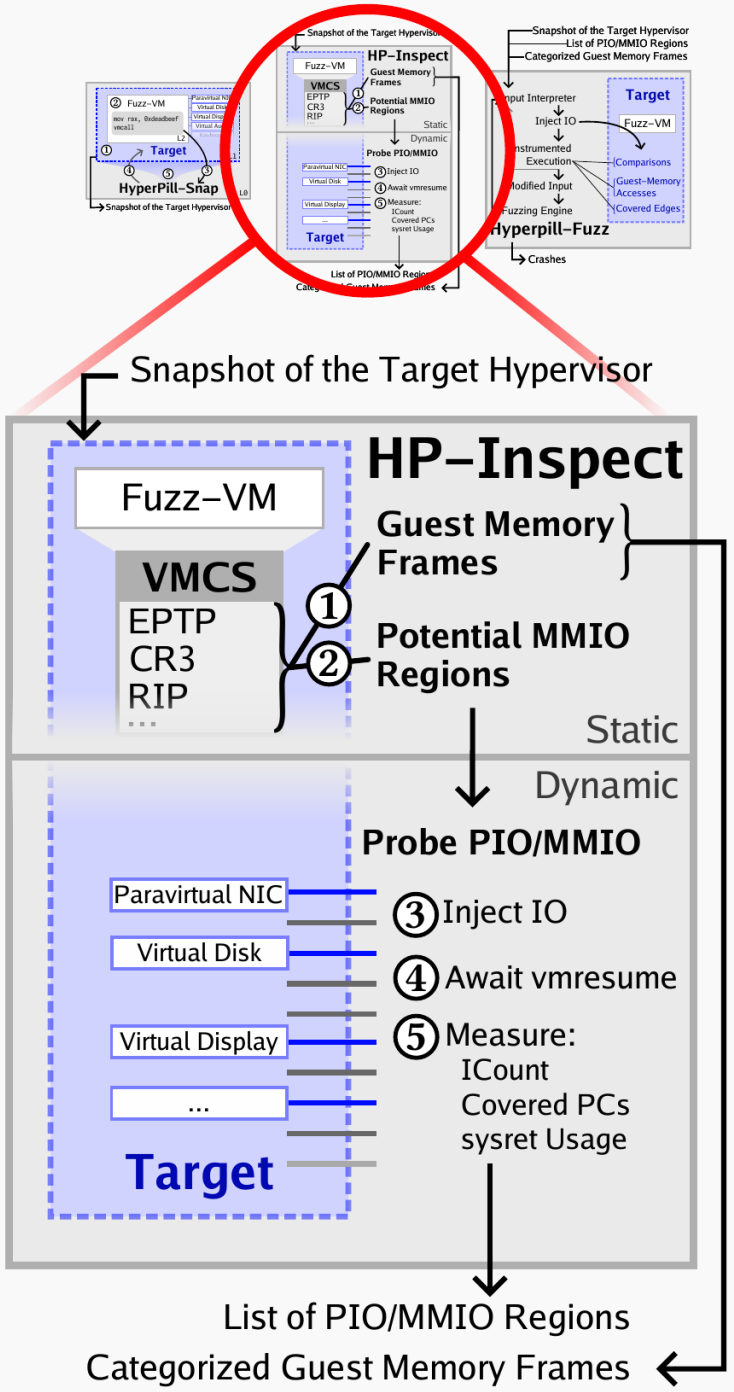
Solution:
Run the hypervisor nested in HyperPill-Snap.

Invoke a special hypercall from the VM to trigger a VM Exit and tell HyperPill-Snap to collect a snapshot.



Step 2: Inspecting the Snapshot

The snapshot contains the entire hypervisor state, including the VMCS the hypervisor sets up for the VM.



Step 2: Inspecting the Snapshot

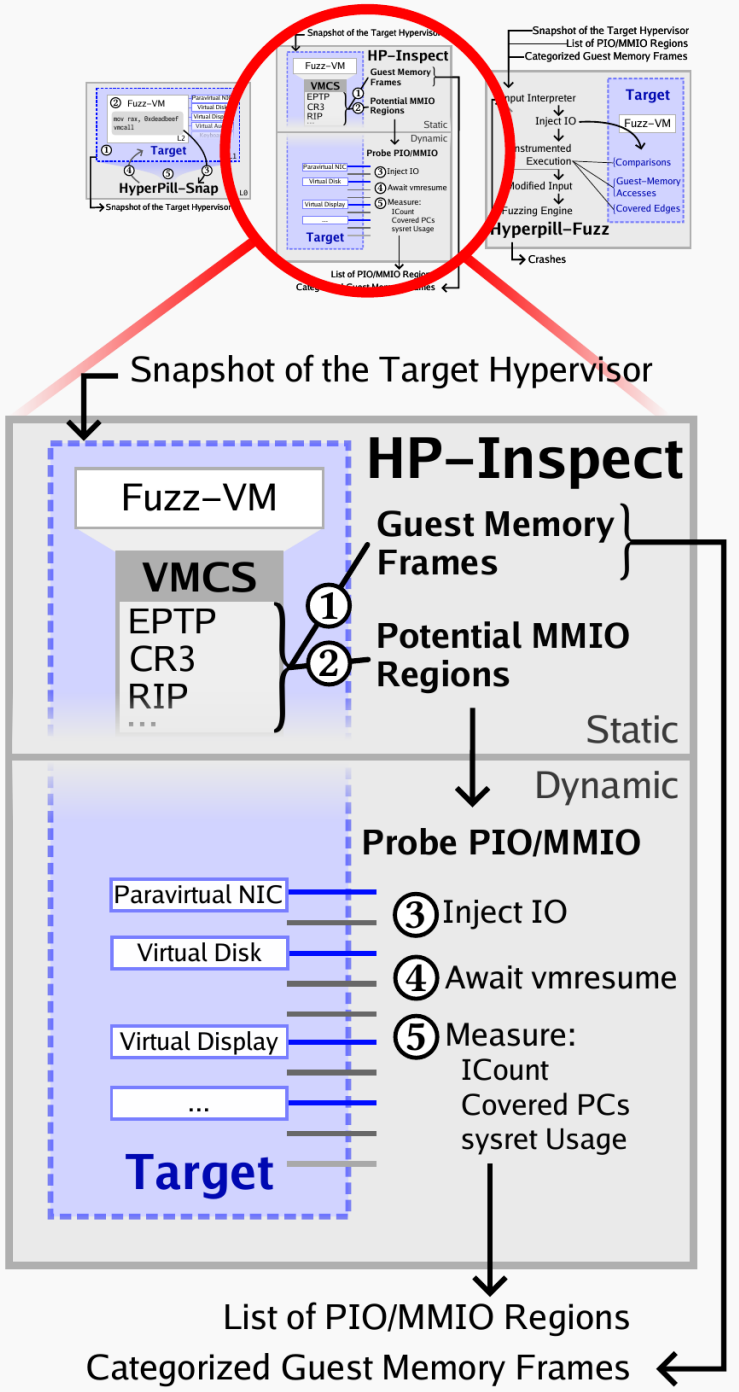
The snapshot contains the entire hypervisor state, including the VMCS the hypervisor sets up for the VM.

VMCS Inspection/Probing to Identify:

MMIO Regions

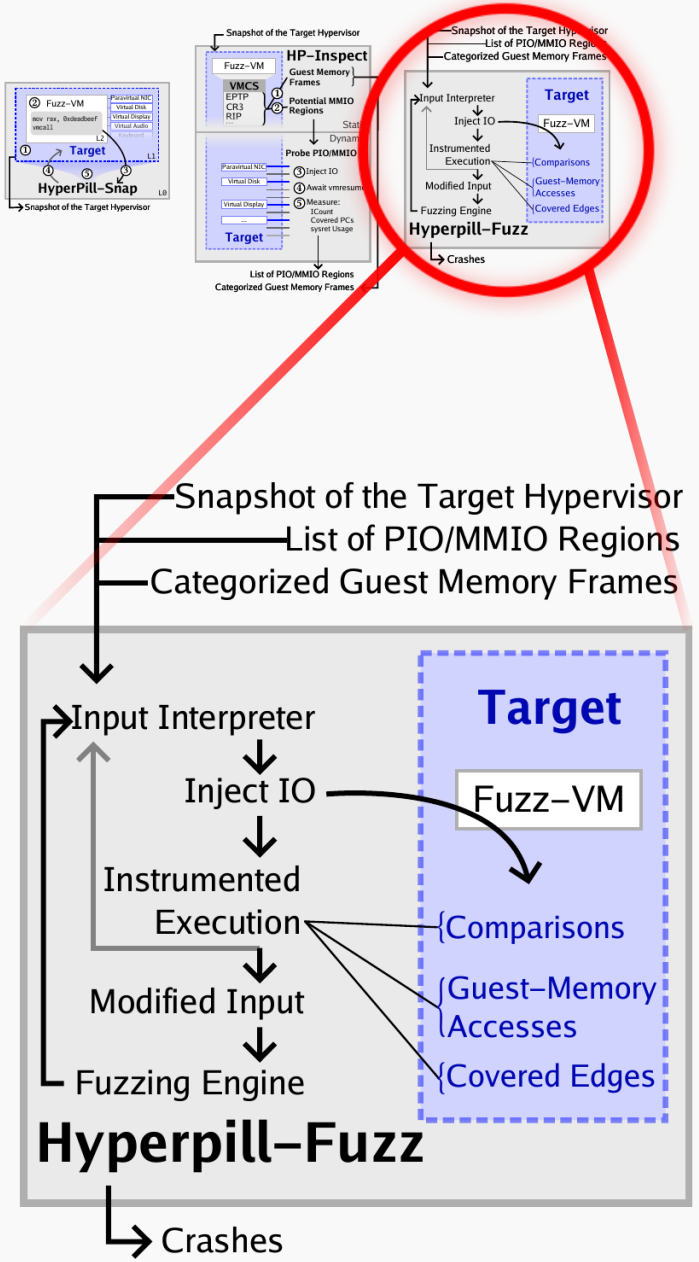
PIO Regions

Memory allocated to the Guest



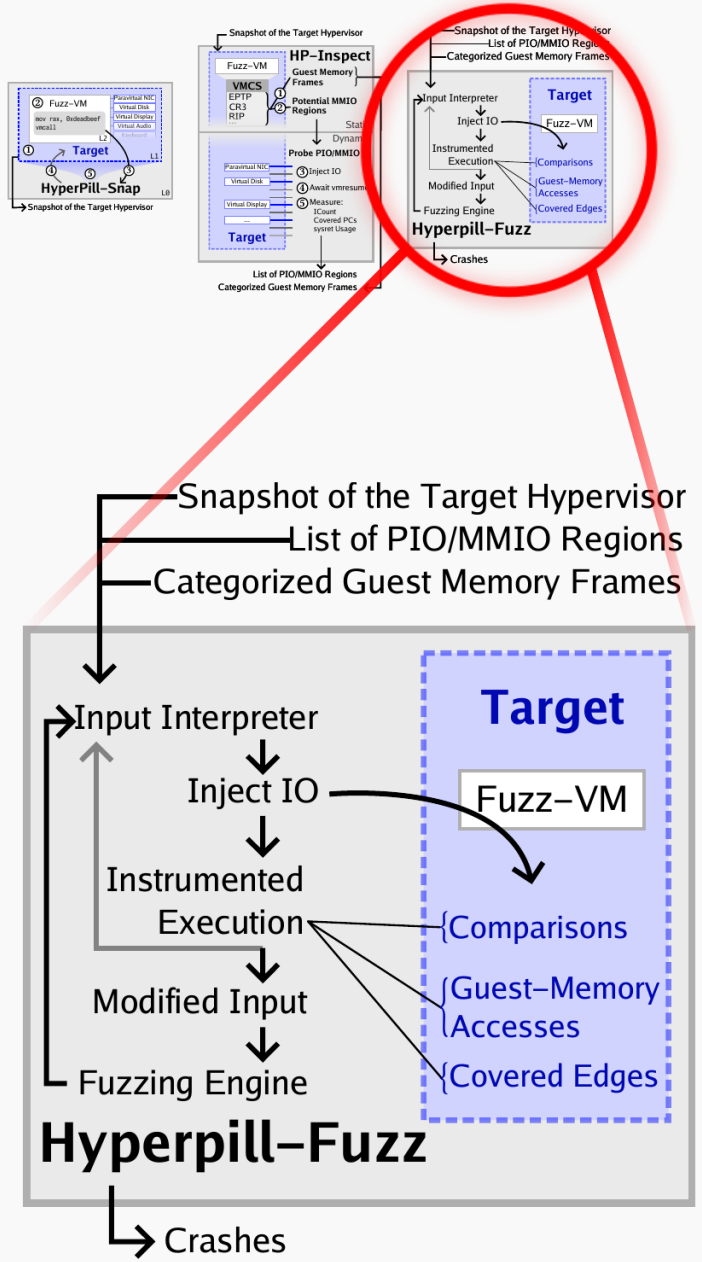
Step 3: Fuzzing

Fuzzer input is a sequence of IO Operations



Step 3: Fuzzing

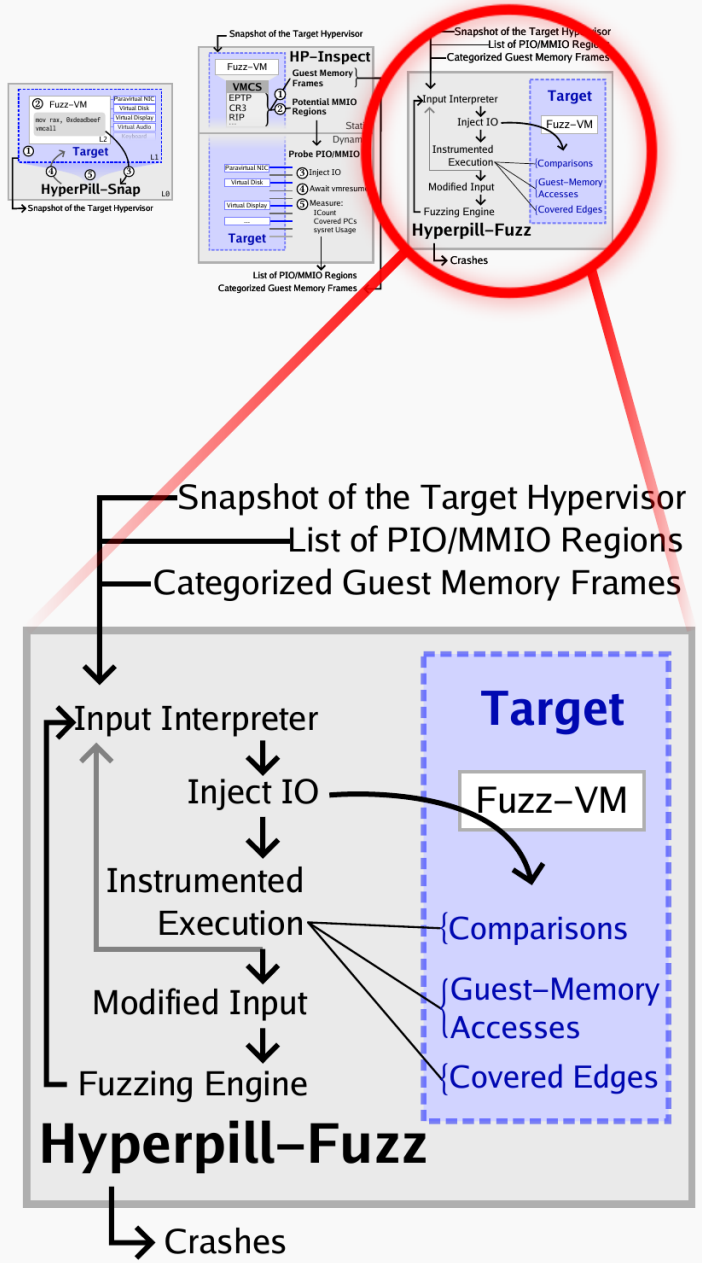
Fuzzer input is a sequence of IO Operations



Step 3: Fuzzing

Fuzzer input is a sequence of IO Operations

Modify VMCS to reflect VMExit for the IO Operation. Resume the Hypervisor Snapshot in an emulator.

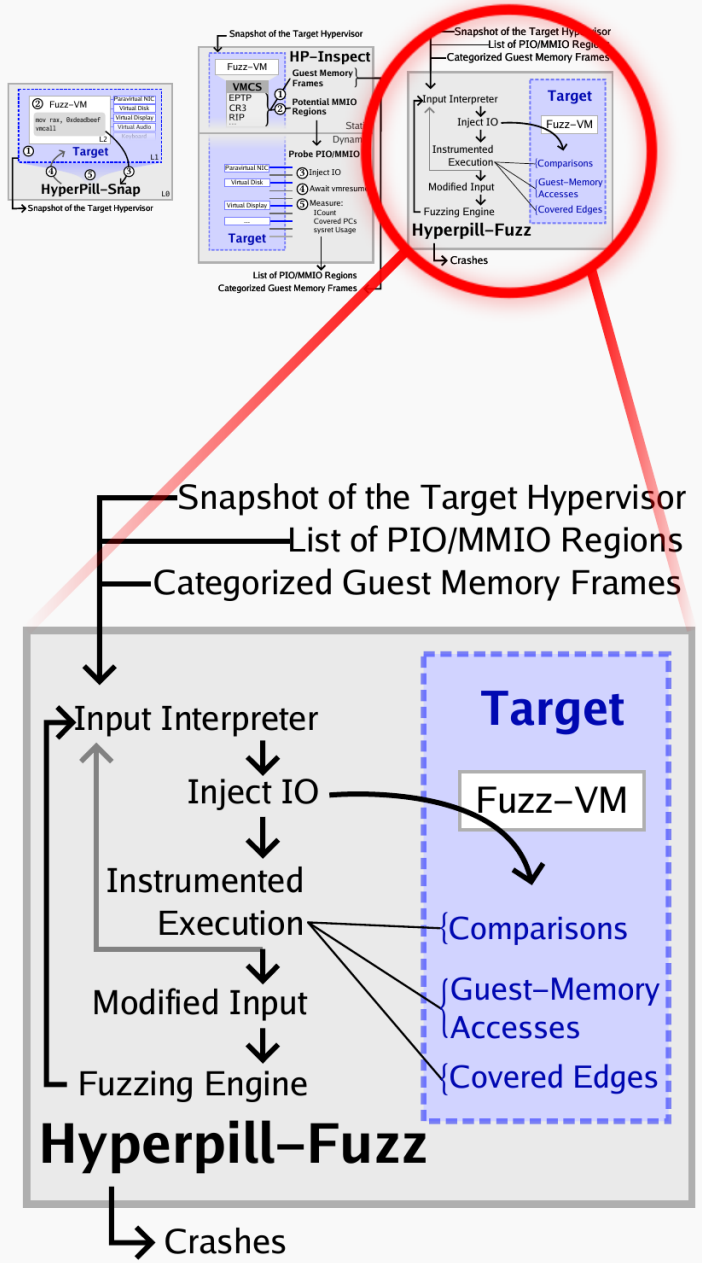


Step 3: Fuzzing

Fuzzer input is a sequence of IO Operations

Modify VMCS to reflect VMExit for the IO Operation. Resume the Hypervisor Snapshot in an emulator.

When the Hypervisor resumes the VM, immediately inject the next VM-Exit



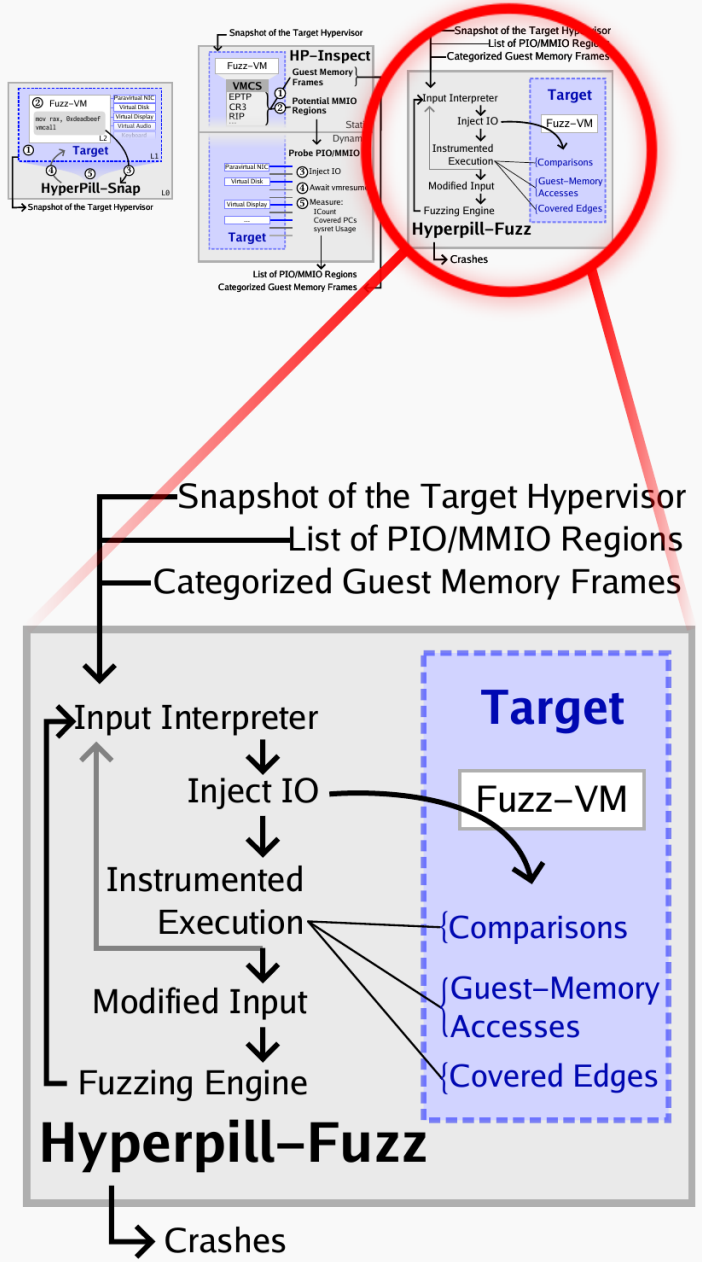
Step 3: Fuzzing

Fuzzer input is a sequence of IO Operations

Modify VMCS to reflect VMExit for the IO Operation. Resume the Hypervisor Snapshot in an emulator.

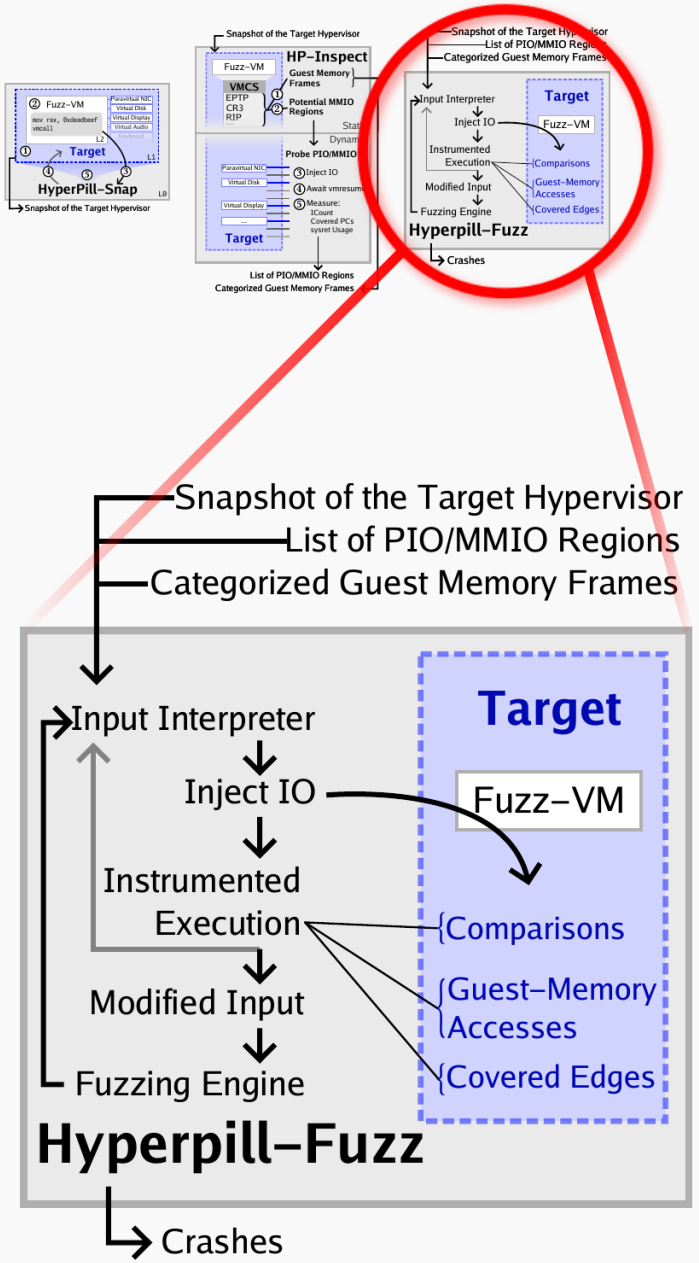
When the Hypervisor resumes the VM, immediately inject the next VM-Exit

PIO ✓ MMIO ✓ MSRs ✓ HyperCalls ✓ DMA ?



Step 3: Fuzzing DMA

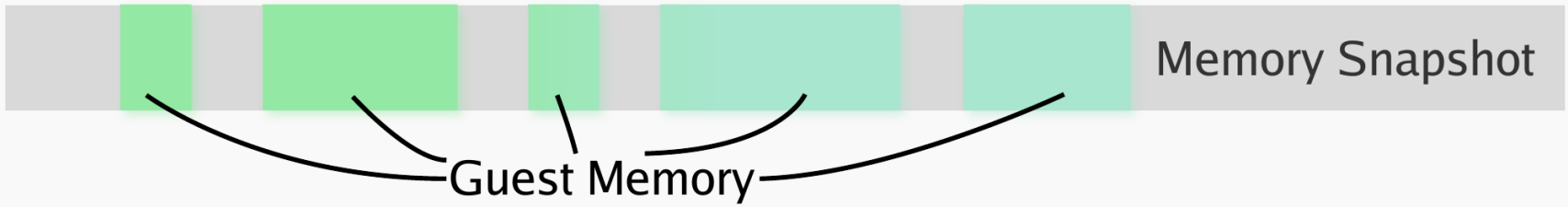
DMA allows hypervisor to read from guest memory at any time while handling VMExits

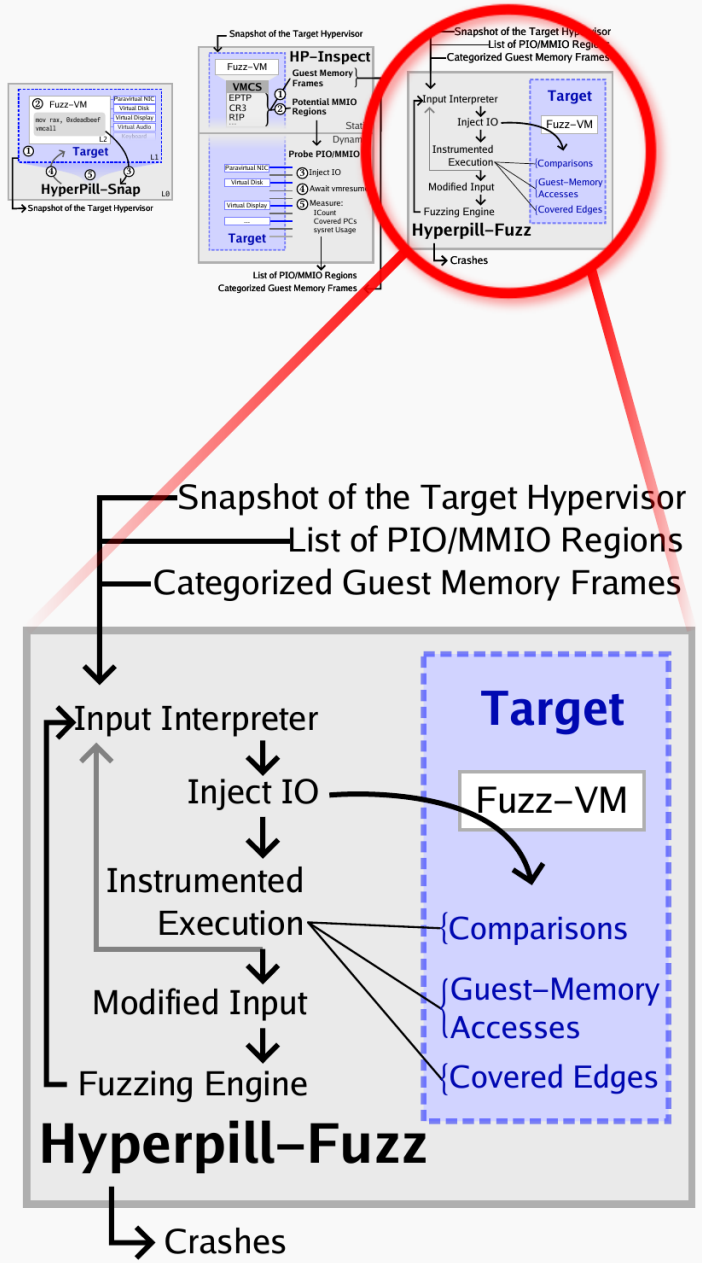


Step 3: Fuzzing DMA

DMA allows hypervisor to read from guest memory at any time while handling VMExits

In step 2, HyperPill identified all Guest Memory pages within the snapshot

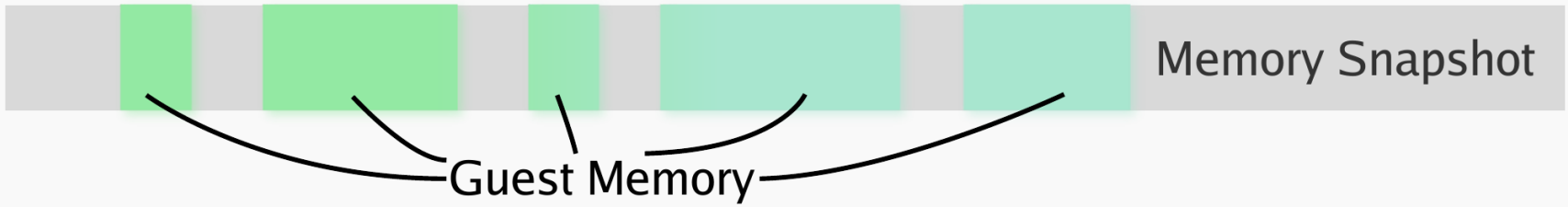




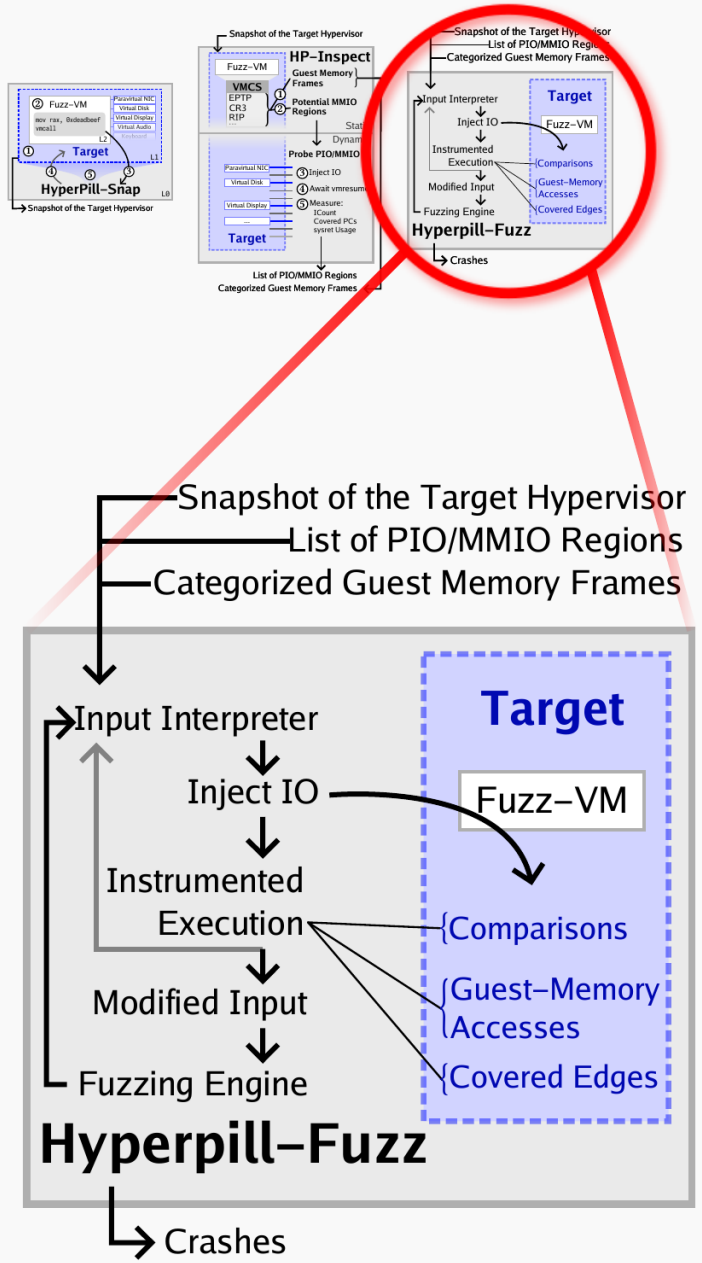
Step 3: Fuzzing DMA

DMA allows hypervisor to read from guest memory at any time while handling VMExits

In step 2, HyperPill identified all Guest Memory pages within the snapshot



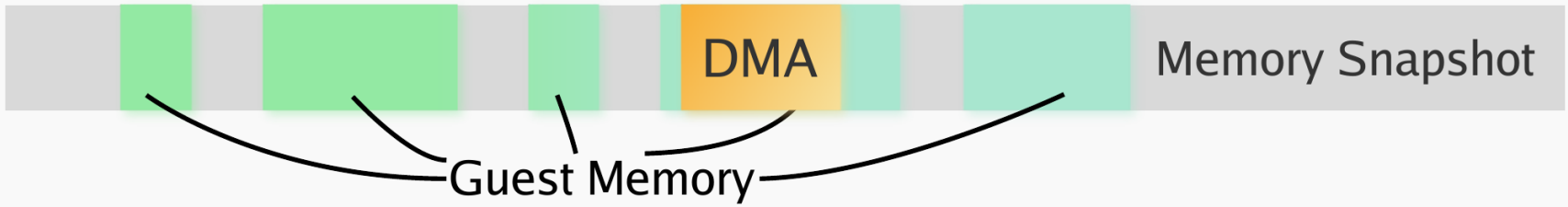
During emulation, if any instruction reads from the Guest Memory, HyperPill fills the memory with fuzzer data



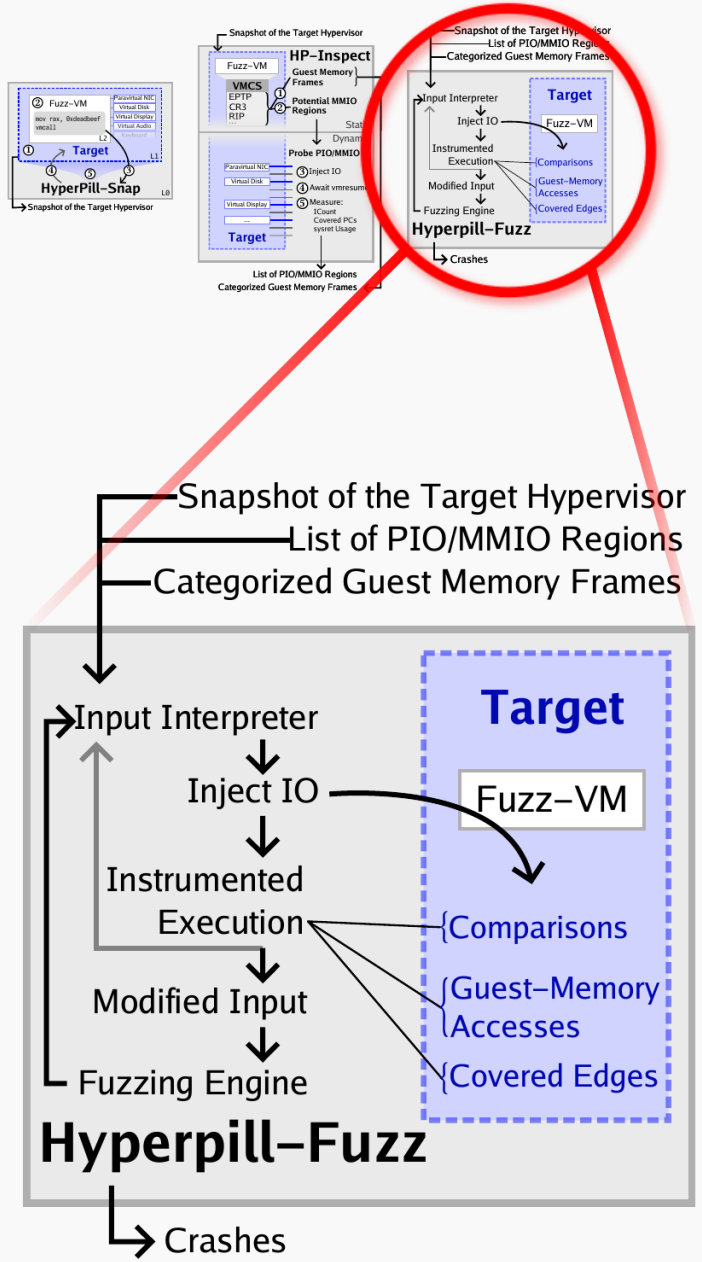
Step 3: Fuzzing DMA

DMA allows hypervisor to read from guest memory at any time while handling VMExits

In step 2, HyperPill identified all Guest Memory pages within the snapshot



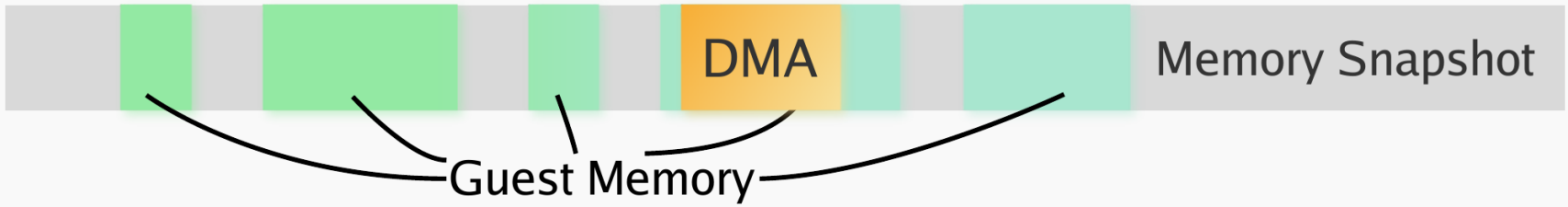
During emulation, if any instruction reads from the Guest Memory, HyperPill fills the memory with fuzzer data



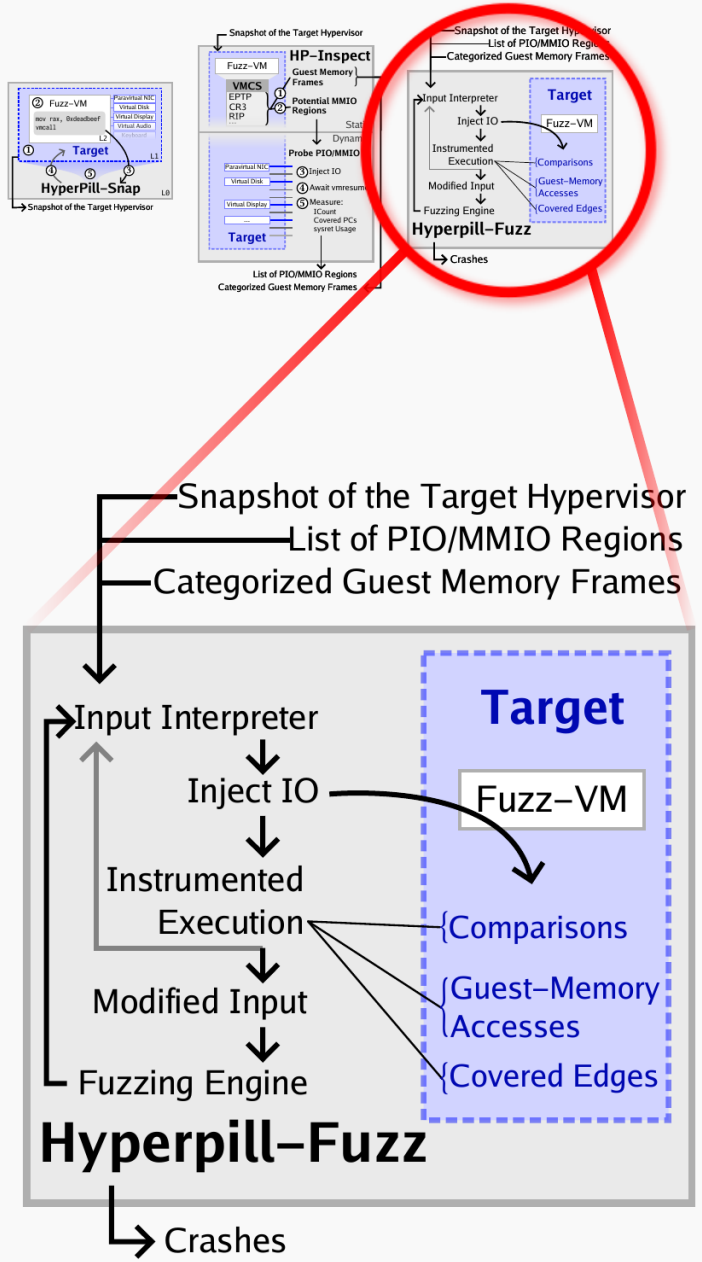
Step 3: Fuzzing DMA

DMA allows hypervisor to read from guest memory at any time while handling VMExits

In step 2, HyperPill identified all Guest Memory pages within the snapshot

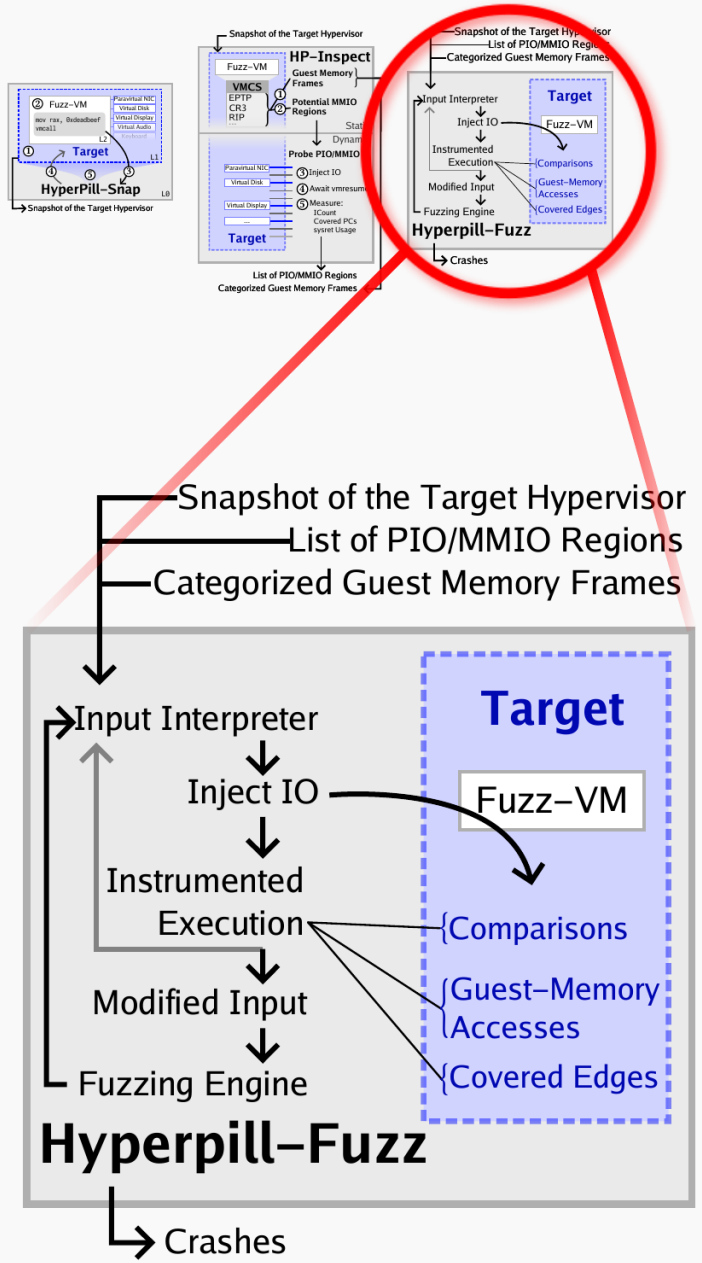


During emulation, if any instruction reads from the Guest Memory, HyperPill fills the memory with fuzzer data



Step 3: Fuzzing DMA

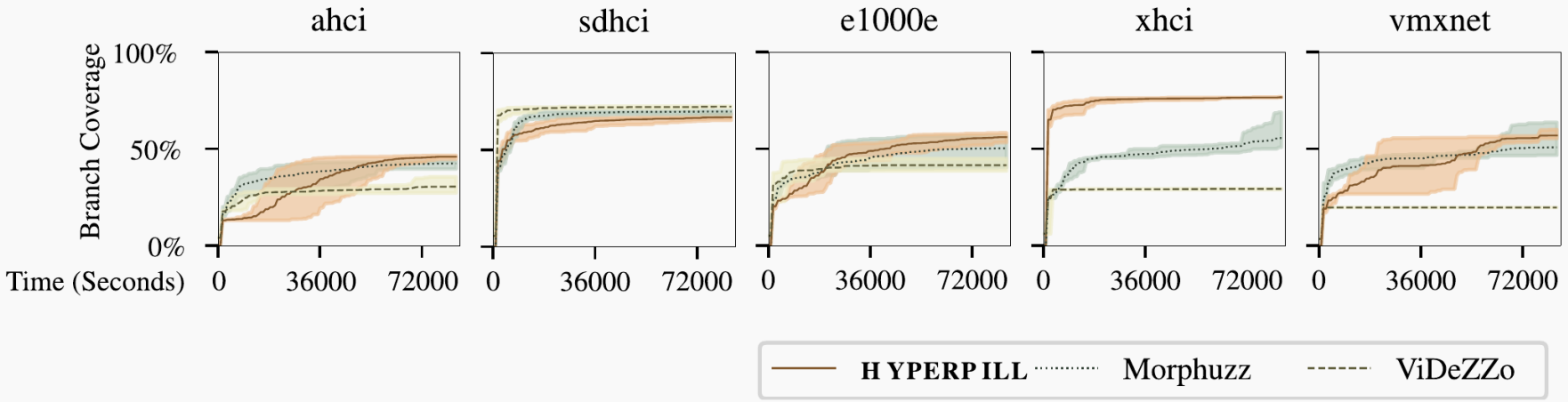
PIO ✓ MMIO ✓ MSRs ✓ HyperCalls ✓ DMA ✓



Results

Three Hypervisors

		Morphuzz		ViDeZZo		HYPERPILL	
		12 Cores 24 Hours					
Device	⊡	Branch Coverage (Executions/Second)					
Block							
ahci	✓	42.43%	(25.68)	30.42%	(562.24)	**45.90%**	(26.18)
nvme		29.12%	(23.82)			**36.44%**	(14.45)
sdhci	✓	69.81%	(22.98)	**72.37%**	(107.22)	66.85%	(32.34)
virtio-scsi	✓	27.96%	(23.83)	11.73%	(217.28)	**48.83%**	(51.68)
Display							
cirrus		88.10%	(19.06)	83.42%	(138.78)	**88.67%**	(32.18)
qxl	✓					**59.68%**	(26.96)
virtio-gpu	✓	24.37%	(26.21)	2.77%	(222.42)	**45.52%**	(36.53)
Networking							
e1000e	✓	50.27%	(24.83)	41.52%	(53.04)	**55.99%**	(42.22)
igb	✓	29.73%	(25.63)			**35.93%**	(60.85)
vmxnet	✓	50.75%	(27.01)	19.64%	(145.73)	**56.89%**	(48.14)
USB							
ehci	✓	73.76%	(24.58)	**74.38%**	(177.08)	73.32%	(10.46)
xhci	✓	55.54%	(28.83)	29.25%	(1061.36)	**76.64%**	(69.26)
Geo. Mean		45.20%	(24.65)	28.00%	(203.07)	55.45%	(33.20)

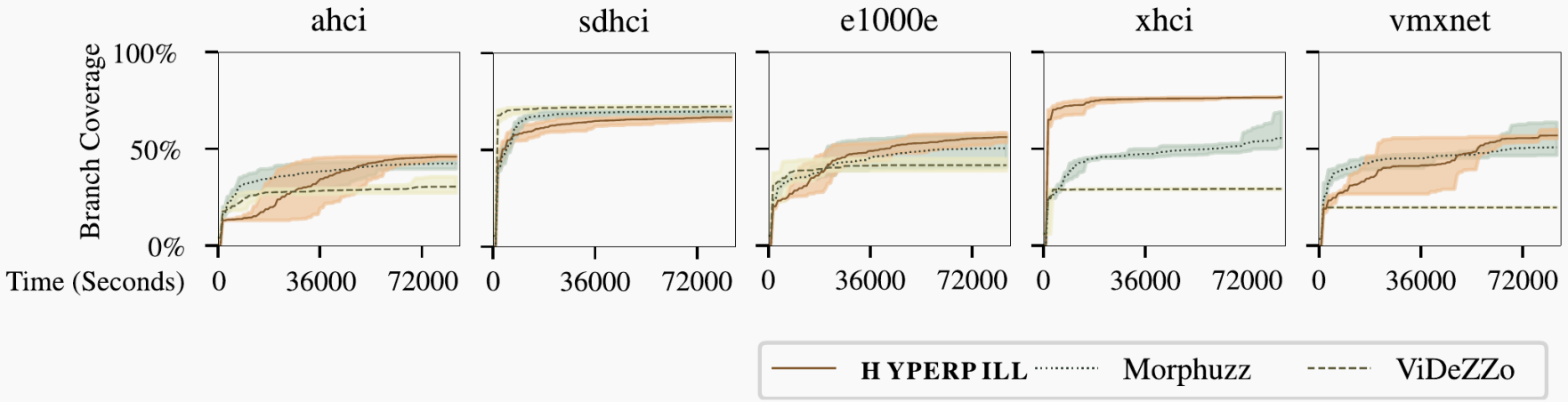


Results

Three Hypervisors

HyperPill achieves higher coverage for 10/12 devices.

		Morphuzz		ViDeZZo		HYPERPILL	
		12 Cores 24 Hours					
Device	⊡	Branch Coverage (Executions/Second)					
Block							
ahci	✓	42.43%	(25.68)	30.42%	(562.24)	**45.90%**	(26.18)
nvme		29.12%	(23.82)			**36.44%**	(14.45)
sdhci	✓	69.81%	(22.98)	**72.37%**	(107.22)	66.85%	(32.34)
virtio-scsi	✓	27.96%	(23.83)	11.73%	(217.28)	**48.83%**	(51.68)
Display							
cirrus		88.10%	(19.06)	83.42%	(138.78)	**88.67%**	(32.18)
qxl	✓					**59.68%**	(26.96)
virtio-gpu	✓	24.37%	(26.21)	2.77%	(222.42)	**45.52%**	(36.53)
Networking							
e1000e	✓	50.27%	(24.83)	41.52%	(53.04)	**55.99%**	(42.22)
igb	✓	29.73%	(25.63)			**35.93%**	(60.85)
vmxnet	✓	50.75%	(27.01)	19.64%	(145.73)	**56.89%**	(48.14)
USB							
ehci	✓	73.76%	(24.58)	**74.38%**	(177.08)	73.32%	(10.46)
xhci	✓	55.54%	(28.83)	29.25%	(1061.36)	**76.64%**	(69.26)
Geo. Mean		45.20%	(24.65)	28.00%	(203.07)	55.45%	(33.20)



Results

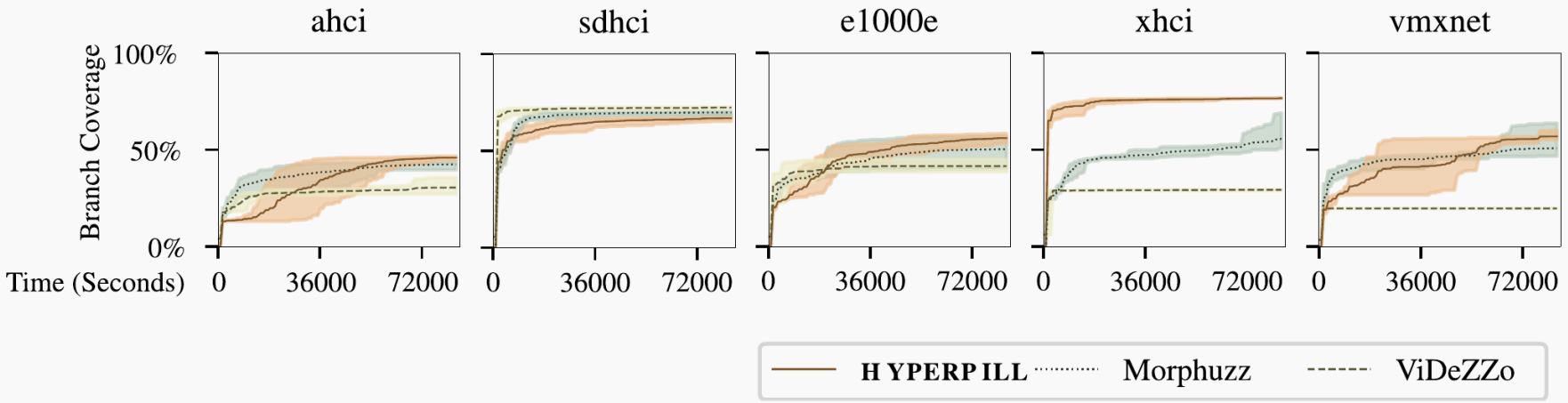
Three Hypervisors

HyperPill achieves higher coverage for 10/12 devices.

Inherent performance penalty due to use of full-system snapshotting and emulation.

Byte-level DMA sensitivity allows better performance for DMA-heavy devices.

		Morphuzz		ViDeZZo		HYPERPILL	
		12 Cores 24 Hours					
Device	⊡	Branch Coverage (Executions/Second)					
Block							
ahci	✓	42.43%	(25.68)	30.42%	(562.24)	**45.90%**	(26.18)
nvme		29.12%	(23.82)			**36.44%**	(14.45)
sdhci	✓	69.81%	(22.98)	**72.37%**	(107.22)	66.85%	(32.34)
virtio-scsi	✓	27.96%	(23.83)	11.73%	(217.28)	**48.83%**	(51.68)
Display							
cirrus		88.10%	(19.06)	83.42%	(138.78)	**88.67%**	(32.18)
qxl	✓					**59.68%**	(26.96)
virtio-gpu	✓	24.37%	(26.21)	2.77%	(222.42)	**45.52%**	(36.53)
Networking							
e1000e	✓	50.27%	(24.83)	41.52%	(53.04)	**55.99%**	(42.22)
igb	✓	29.73%	(25.63)			**35.93%**	(60.85)
vmxnet	✓	50.75%	(27.01)	19.64%	(145.73)	**56.89%**	(48.14)
USB							
ehci	✓	73.76%	(24.58)	**74.38%**	(177.08)	73.32%	(10.46)
xhci	✓	55.54%	(28.83)	29.25%	(1061.36)	**76.64%**	(69.26)
Geo. Mean		45.20%	(24.65)	28.00%	(203.07)	55.45%	(33.20)



```
/* map PRDT */
if (!(prdt = dma_memory_map(ad->hba->as, prdt_addr, &prdt_len,
                            DMA_DIRECTION_TO_DEVICE,
                            MEMTXATTRS_UNSPECIFIED))){
    trace_ahci_populate_sglist_no_map(ad->hba, ad->port_no);
    return -1;
}
/* Get entries in the PRDT, init a qemu sglist accordingly */
if (prdtl > 0) {
    AHCI_SG *tbl = (AHCI_SG *)prdt;

    for (i = 0; i < prdtl; i++) {
        tbl_entry_size = prdt_tbl_entry_size(&tbl[i]);
        if (offset < (sum + tbl_entry_size)) {
            off_idx = i
```

Source-Based DMA Hooking

```
/* map PRDT */
if (!(prdt = dma_memory_map(ad->hba->as, prdt_addr, &prdt_len,
                            DMA_DIRECTION_TO_DEVICE,
                            MEMTXATTRS_UNSPECIFIED))){
    trace_ahci_populate_sglist_no_map(ad->hba, ad->port_no);
    return -1;
}
/* Get entries in the PRDT, init a qemu sglist accordingly */
if (prdtl > 0) {
    AHCI_SG *tbl = (AHCI_SG *)prdt;

    for (i = 0; i < prdtl; i++) {
        tbl_entry_size = prdt_tbl_entry_size(&tbl[i]);
        if (offset < (sum + tbl_entry_size)) {
            off_idx   i
```

Source-Based DMA Hooking

```
leaq    (%r15,%r12), %r14
incq    %r14
movq    %r14, %rdi
movq    %rbx, %rsi
```

HyperPill DMA Hooking

Bugs

QEMU

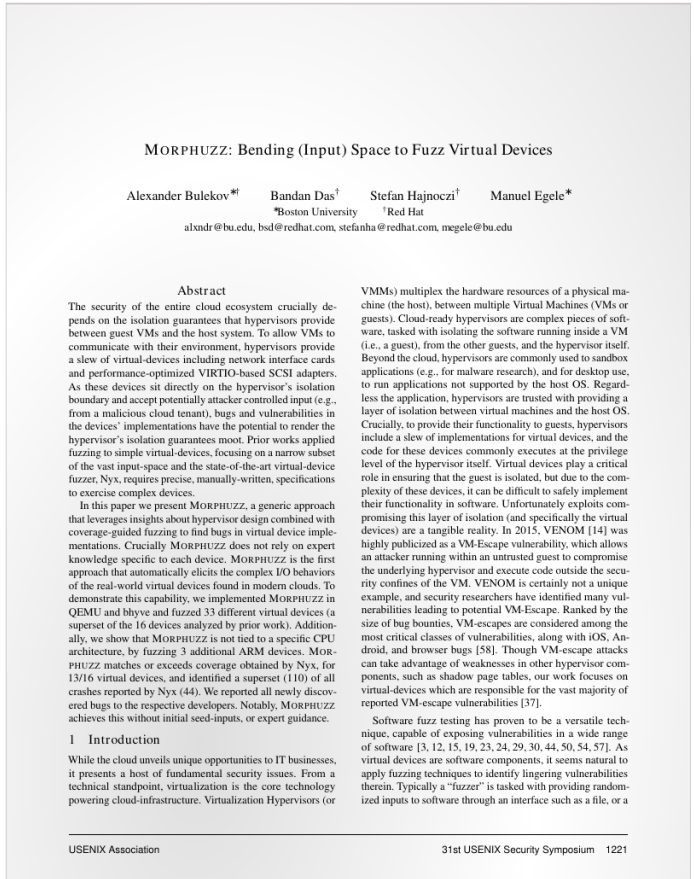
- Arbitrary memory-access in e1000e_start_xmit
- Heap-overflow in usb_mouse_poll
- Heap-overflow in virtqueue_alloc_element
- Heap-overflow in qxl_cookie_new
- Heap-overflow in igb_tx_pkt_switch
- Out-of-bounds memory access in nvme_process_sq
- Out-of-bounds memory access in nvme_io_mgmt_send
- DoS via arbitrary-sized allocation in qxl
- DoS via arbitrary-sized allocation in virtio_gpu
- DoS in process_ncq_command
- DoS in icmp_input

Hyper-V

- Heap-corruption in EthernetCard::HandleTransmitSetupFrame
- Abort in EthernetCard::PollForTransmitDataTimer
- Abort after IdeChannel::EnlightenedHddCommand
- EthernetCard::SetupEthernetCardModeFromRegisters
- Out-of-bounds write in GuestStateAccess::SetDeviceInfo
- Abort after PitDevice::NotifyIoPortRead
- Abort in I8042Device::HandleCommand
- Abort after HvCallDetachDevice
- Abort after HvCallGetGpaPagesAccessState

macOS Virtualization Framework

- Memory-privilege violation in xHCI
- Out-of-bounds write in virtio-gpu
- Out-of-bounds write in virtio-audio
- Out-of-bounds access in virtio-block
- Out-of-bounds access in virtio-console
- Out-of-bounds access in virtio-net



MORPHUZZ: Bending (Input) Space to Fuzz Virtual Devices

Alexander Bulekov*† Bandan Das† Stefan Hajnoczi† Manuel Egele*
*Boston University †Red Hat
alxndr@bu.edu, bsd@redhat.com, stefanha@redhat.com, megele@bu.edu

Abstract

The security of the entire cloud ecosystem crucially depends on the isolation guarantees that hypervisors provide between guest VMs and the host system. To allow VMs to communicate with their environment, hypervisors provide a slew of virtual-devices including network interface cards and performance-optimized VIRTIO-based SCSI adapters. As these devices sit directly on the hypervisor's isolation boundary and accept potentially attacker controlled input (e.g., from a malicious cloud tenant), bugs and vulnerabilities in the devices' implementations have the potential to render the hypervisor's isolation guarantees moot. Prior works applied fuzzing to simple virtual-devices, focusing on a narrow subset of the vast input-space and the state-of-the-art virtual-device fuzzer, Nyx, requires precise, manually-written, specifications to exercise complex devices.

In this paper we present MORPHUZZ, a generic approach that leverages insights about hypervisor design combined with coverage-guided fuzzing to find bugs in virtual device implementations. Crucially MORPHUZZ does not rely on expert knowledge specific to each device. MORPHUZZ is the first approach that automatically elicits the complex I/O behaviors of the real-world virtual devices found in modern clouds. To demonstrate this capability, we implemented MORPHUZZ in QEMU and bhyve and fuzzed 33 different virtual devices (a superset of the 16 devices analyzed by prior work). Additionally, we show that MORPHUZZ is not tied to a specific CPU architecture, by fuzzing 3 additional ARM devices. MORPHUZZ matches or exceeds coverage obtained by Nyx, for 13/16 virtual devices, and identified a superset (110) of all crashes reported by Nyx (44). We reported all newly discovered bugs to the respective developers. Notably, MORPHUZZ achieves this without initial seed-inputs, or expert guidance.

1 Introduction

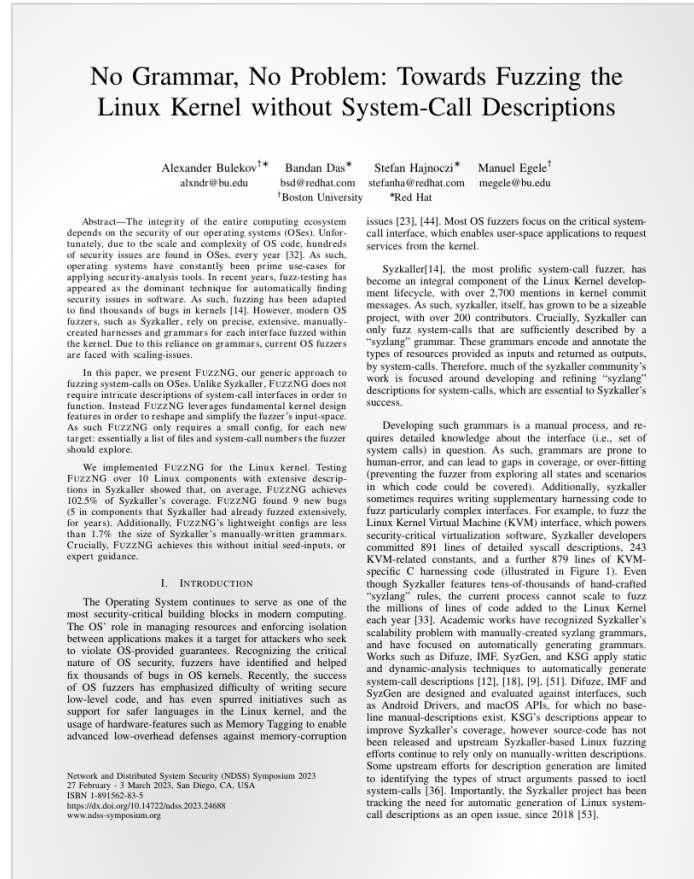
While the cloud unveils unique opportunities to IT businesses, it presents a host of fundamental security issues. From a technical standpoint, virtualization is the core technology powering cloud-infrastructure. Virtualization Hypervisors (or VMMs) multiplex the hardware resources of a physical machine (the host), between multiple Virtual Machines (VMs or guests). Cloud-ready hypervisors are complex pieces of software, tasked with isolating the software running inside a VM (i.e., a guest), from the other guests, and the hypervisor itself. Beyond the cloud, hypervisors are commonly used to sandbox applications (e.g., for malware research), and for desktop use, to run applications not supported by the host OS. Regardless the application, hypervisors are trusted with providing a layer of isolation between virtual machines and the host OS. Crucially, to provide their functionality to guests, hypervisors include a slew of implementations for virtual devices, and the code for these devices commonly executes at the privilege level of the hypervisor itself. Virtual devices play a critical role in ensuring that the guest is isolated, but due to the complexity of these devices, it can be difficult to safely implement their functionality in software. Unfortunately exploits compromising this layer of isolation (and specifically the virtual devices) are a tangible reality. In 2015, VENOM [14] was highly publicized as a VM-Escape vulnerability, which allows an attacker running within an untrusted guest to compromise the underlying hypervisor and execute code outside the security confines of the VM. VENOM is certainly not a unique example, and security researchers have identified many vulnerabilities leading to potential VM-Escape. Ranked by the size of bug bounties, VM-escapes are considered among the most critical classes of vulnerabilities, along with iOS, Android, and browser bugs [58]. Though VM-escape attacks can take advantage of weaknesses in other hypervisor components, such as shadow page tables, our work focuses on virtual-devices which are responsible for the vast majority of reported VM-escape vulnerabilities [37].

Software fuzz testing has proven to be a versatile technique, capable of exposing vulnerabilities in a wide range of software [3, 12, 15, 19, 23, 24, 29, 30, 44, 50, 54, 57]. As virtual devices are software components, it seems natural to apply fuzzing techniques to identify lingering vulnerabilities therein. Typically a "fuzzer" is tasked with providing randomized inputs to software through an interface such as a file, or a

MORPHUZZ

Bending (Input) Space to Fuzz Virtual Devices

USENIX Security 2022



No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions

Alexander Bulekov†* Bandan Das* Stefan Hajnoczi* Manuel Egele†
alxndr@bu.edu bsd@redhat.com stefanha@redhat.com megele@bu.edu
†Boston University *Red Hat

Abstract—The integrity of the entire computing ecosystem depends on the security of our operating systems (OSes). Unfortunately, due to the scale and complexity of OS code, hundreds of security issues are found in OSes, every year [32]. As such, operating systems have constantly been prime use-cases for applying security-analysis tools. In recent years, fuzz-testing has appeared as the dominant technique for automatically finding security issues in software. As such, fuzzing has been adapted to find thousands of bugs in kernels [14]. However, modern OS fuzzers, such as Syzkaller, rely on precise, extensive, manually-created harnesses and grammars for each interface fuzzed within the kernel. Due to this reliance on grammars, current OS fuzzers are faced with scaling-issues.

In this paper, we present FUZZNG, our generic approach to fuzzing system-calls on OSes. Unlike Syzkaller, FUZZNG does not require intricate descriptions of system-call interfaces in order to function. Instead FUZZNG leverages fundamental kernel design features in order to reshape and simplify the fuzzer's input-space. As such FUZZNG only requires a small config, for each new target: essentially a list of files and system-call numbers the fuzzer should explore.

We implemented FUZZNG for the Linux kernel. Testing FUZZNG over 10 Linux components with extensive descriptions in Syzkaller showed that, on average, FUZZNG achieves 102.5% of Syzkaller's coverage. FUZZNG found 9 new bugs (5 in components that Syzkaller had already fuzzed extensively, for years). Additionally, FUZZNG's lightweight configs are less than 1.7% the size of Syzkaller's manually-written grammars. Crucially, FUZZNG achieves this without initial seed-inputs, or expert guidance.

I. INTRODUCTION

The Operating System continues to serve as one of the most security-critical building blocks in modern computing. The OS' role in managing resources and enforcing isolation between applications makes it a target for attackers who seek to violate OS-provided guarantees. Recognizing the critical nature of OS security, fuzzers have identified and helped fix thousands of bugs in OS kernels. Recently, the success of OS fuzzers has emphasized difficulty of writing secure low-level code, and has even spurred initiatives such as support for safer languages in the Linux kernel, and the usage of hardware-features such as Memory Tagging to enable advanced low-overhead defenses against memory-corruption issues [23], [44]. Most OS fuzzers focus on the critical system-call interface, which enables user-space applications to request services from the kernel.

Syzkaller[14], the most prolific system-call fuzzer, has become an integral component of the Linux Kernel development lifecycle, with over 2,700 mentions in kernel commit messages. As such, syzkaller, itself, has grown to be a sizeable project, with over 200 contributors. Crucially, Syzkaller can only fuzz system-calls that are sufficiently described by a "syzlang" grammar. These grammars encode and annotate the types of resources provided as inputs and returned as outputs, by system-calls. Therefore, much of the syzkaller community's work is focused around developing and refining "syzlang" descriptions for system-calls, which are essential to Syzkaller's success.

Developing such grammars is a manual process, and requires detailed knowledge about the interface (i.e., set of system calls) in question. As such, grammars are prone to human-error, and can lead to gaps in coverage, or over-fitting (preventing the fuzzer from exploring all states and scenarios in which code could be covered). Additionally, syzkaller sometimes requires writing supplementary harnessing code to fuzz particularly complex interfaces. For example, to fuzz the Linux Kernel Virtual Machine (KVM) interface, which powers security-critical virtualization software, Syzkaller developers committed 891 lines of detailed syscall descriptions, 243 KVM-related constants, and a further 879 lines of KVM-specific C harnessing code (illustrated in Figure 1). Even though Syzkaller features tens-of-thousands of hand-crafted "syzlang" rules, the current process cannot scale to fuzz the millions of lines of code added to the Linux Kernel each year [33]. Academic works have recognized Syzkaller's scalability problem with manually-created syzlang grammars, and have focused on automatically generating grammars. Works such as Difuze, IMF, SyzGen, and KSG apply static and dynamic-analysis techniques to automatically generate system-call descriptions [12], [18], [9], [51]. Difuze, IMF and SyzGen are designed and evaluated against interfaces, such as Android Drivers, and macOS APIs, for which no baseline manual-descriptions exist. KSG's descriptions appear to improve Syzkaller's coverage, however source-code has not been released and upstream Syzkaller-based Linux fuzzing efforts continue to rely only on manually-written descriptions. Some upstream efforts for description generation are limited to identifying the types of struct arguments passed to ioctl system-calls [36]. Importantly, the Syzkaller project has been tracking the need for automatic generation of Linux system-call descriptions as an open issue, since 2018 [53].

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24688
www.ndss-symposium.org

FUZZNG

Towards Fuzzing the Linux Kernel without System-Call Descriptions

NDSS 2023

Reshape hypervisors by modifying the CPU virtualization interface
No modification to hypervisors code needed!

Fuzz *any* hypervisor across its PIO, MMIO, DMA, and Hypercall interfaces

More precise than source-level reshaping

HyperPill

https://github.com/HexHive/HyperPill

RSAC 2025 Conference

Many Voices.
One Community.

Thank You

For more information:

https://dl.acm.org/doi/10.5555/3698900.3698952

https://github.com/HexHive/HyperPill

https://www.linkedin.com/in/a1xndr/